Evidence-based Software Engineering

based on the publicly available data

Derek M. Jones

Contents

Read me 1st

The aim when writing this book was to discuss what is currently known about software engineering, based on an analysis of all publicly available software engineering data.[i]

This aim is not as ambitious as it sounds, because your author knew from experience that there is not much data publicly available. Researchers in software engineering tend to focus on vanity interests, and write papers intended to induce mathematical orgasms, i.e., not research that might uncover something useful to industry based on experimental evidence.

If data relating to a topic is not publicly available, the topic is not discussed. Adhering to this rule has led to a somewhat disjoint discussion, although it does vividly highlight the almost non-existent evidence for theories of software development in general.

The material continues to be disjoint because data discovery has been a disjoint leap into the unknown. The organization has shifted as newly discovered data has changed the relationships between what has already been discussed.

The number of publicly available datasets (620+, roughly) turned out to be three to four times more than I had thought was available.

The intended audience is software developers and their managers.

The material assumes the reader is fluent in at least one programming language and has some basic mathematical skills, e.g., knows a little about probability, permutations, and the idea of measurements containing some amount of error. Developers are assumed to be casual users of statistics who don't want to spend time learning lots of mathematics, they want to use the techniques, not implement them.

It is assumed that developer time is expensive and computer time is cheap. Where possible a single, general, statistical technique is described, along with a single way of coding something in `R` is used. This minimal, but general approach focuses on what developers need to know, and the price paid is that the code that does the analysis may be slower (in many cases the performance slowdown is unlikely to be noticeable).

What topics are part of software engineering, and which data analysis techniques are generally applicable to software developers?

This book takes an inclusive approach, awaiting practical experience to winnow the less widely applicable topics and data analysis techniques.

Your author's life would have been easier if it had been possible to point readers at an existing book to learn about statistics. Unfortunately existing statistics books are not suitable, for reasons that include:

- they contain too much implementation detail. Developers want to use the techniques, not implement them.
- they target the largest market for introductory statistics books, the social sciences. The characteristics of the data encountered in social sciences are very different from the data encountered in software engineering, e.g., the Normal distribution is commonly encountered in social science data but is not that common in software engineering data, while the exponential distribution is common in software engineering and much less common in the social sciences.

If you know of some interesting software engineering data not discussed here, please tell me where I can obtain a copy.

All the code and data can be downloaded at: github.com/Derek-Jones/ESEUR-code-data

[i] This is a beta release, and there are roughly another 20 datasets waiting to be analysed and discussed.

Acknowledgements

Thanks to the researchers who made their data available on the web, or responded to my email request by sending me a copy of their data. Without this data, there would be no book.

In several dozen cases WebPlotDigitizer was used to extract data from plots in published papers, where the original data was not available.

Various website played an essential role in locating papers, blog posts and data. These include: Google search, Google Scholar, ResearchGate, Semantic Scholar, Archive.org, and the Defense Technical Information Center.

Thanks to the maintainers of R, CRAN, and the authors of the several hundred packages used to analyse data.

The text was written using Asciidoc, with LuaLaTex generating the pdf. Language Tool was used for grammar checking.

Thanks to Github for hosting the book's code+data.

Chapter 1

Introduction

Software systems and their host, the electronic computer, began[308, 602, 641, 1766] infiltrating the human ecosystem 70+ years ago.[i] During this period the price of computer equipment continually declined, averaging 17.5% per year,[1823] an economic firestorm that devoured existing practices; software systems are apex predators. Figure 1.1 shows the continual fall in the cost of compute operations, however, without cheap mass storage computing would be a niche market; the continual reduction in the cost of storage generated increasing economic incentives for employing the cheap processing power; see figure 1.2. The pattern of hardware performance improvements (and shortage of trained personnel) were appreciated almost from the beginning.[739]

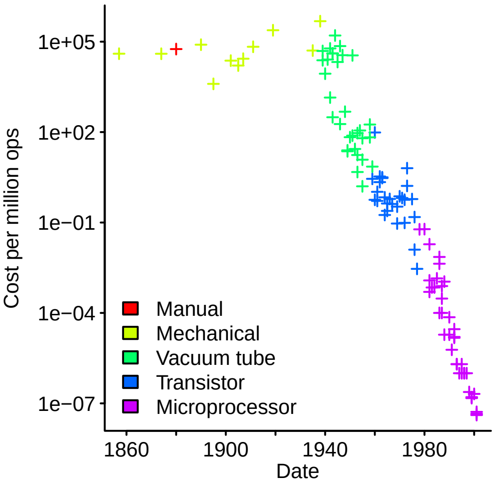


Figure 1.1: Total cost of one million computing operations over time. Data from Nordhaus.[1369] Github–Local

A shift in perception, from computers as calculating machines,[733] to computing platforms responding in real-time, to multiple independent users, created new problem solving opportunities, e.g., responding in real-time to multiple users; figure 1.3 shows the growth of US based systems capable of sharing cpu time between multiple users.[160]

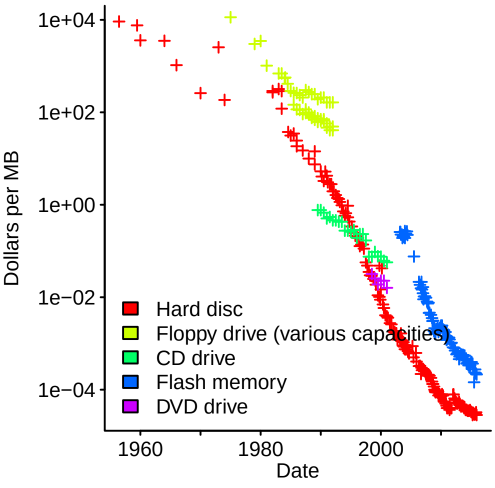


Figure 1.2: Storage cost, in US dollars per Mbyte, of mass market technologies over time. Data from McCallum,[1214] floppy and CD-ROM data kindly provided by Davis.[439] Github–Local

Practical software systems are a combination of hardware and software; with each having very different production economics. The cost of designing and building the first instance of hardware is small compared to the cost of reproducing it in quantity; the cost of designing and building the first instance of software is huge compared to the cost of reproducing it in quantity.[ii]

Software is a product of human cognitive labor, applying the skills and know-how acquired through practical experience. Software engineering started as, and has remained a craft activity.

Craft activities are built upon the experience of what has been found to work in practice; when doing something completely new, practical experience is an effective technique for acquiring the skills and know-how of a new craft. Some activities have progressed to include an engineering/scientific approach, based on theories that model the underlying processes; the theories are formulated and validated using evidence obtained from experiments and measurements of events in the world; the benefits of such an approach include greater control and predictability of the creation process (e.g., costs are reduced by reducing wasted resources).

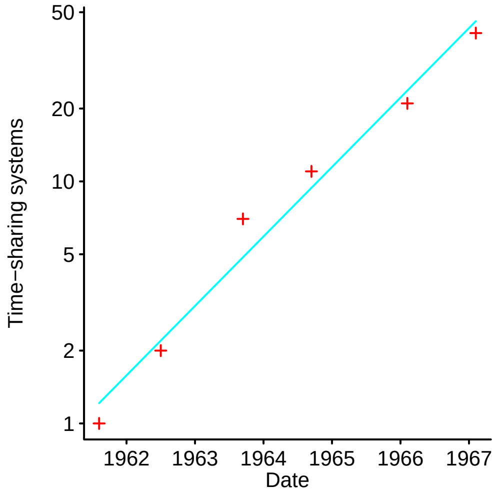


Figure 1.3: Growth of time-sharing systems available in the US, with fitted regression line. Data extracted from Glauthier.[677] Github–Local

Theories are a source of free information, i.e., they provide a basis for understanding the workings of a system, and for making good enough predictions.

Very little progress has been made towards creating practical theories capable of supporting an engineering/scientific approach to software development. Many vanity theories have been proposed (i.e., they are based on the personal beliefs of researchers, rather than evidence obtained from experiments and measurements of software development); academic software engineering research is a backwater with a tenuous connection to practical software development.

The demand for people capable of solving the problems involved in building complex software systems has drawn talent away from research.

[i] The verb "to program" was first used in its modern sense in 1946.[732] This book focuses on computers that operate by controlling the flow of electrons (i.e., liquid based flow control computers are not discussed[6]).

[ii] It is a mistake to compare factory production, which makes copies of hardware products, with software production, which creates a new product that can be copied at almost zero cost.

Research into software engineering is poorly funded, compared to projects where, for instance, hundreds of millions are spent on sending spacecraft to take snapshots of other planets, and telescopes to photograph very far away objects.

Software systems is a sellers' market, the customer will pay what it takes because the potential benefits are so much greater than the costs. In a sellers' market, vendors don't need to lobby government to fund research into reducing their costs. The benefits of an engineering/scientific approach to development, are primarily reaped by customers (e.g., lower costs, more timely delivery, and fewer faults experienced); those who develop software systems are not motivated to invest in change when customers are willing to continue paying for systems developed using craft practices (unless they are also the customer).

The continued displacement of existing systems and practices has kept the focus of practice on development (of new systems and major enhancements to existing systems). A change in the focus of practice to infrastructure[1837] has to await longer term stability.

The primary goal of this book is to discuss all the publicly available software engineering data,[927] with software engineering treated as an economically motivated cognitive activity (occurring within one or more cultures). A topic is only discussed if measurement data is publicly available to ground the discussion. Keeping to this requirement means that readers are likely to be dismayed at the scant coverage of many topics of importance in software engineering. The intended audience are those involved in building software systems.

The focus is on understanding, not prediction. Those involved in building software systems want to control the process. Control requires understanding; an understanding of the many processes involved in building software systems is the goal of software engineering research.

Evidence (i.e., experimental and measurement data) is analysed using statistics, with statistical techniques being wielded as weaponised pattern recognition; those seeking discussions written in the style of a stimulant for mathematical orgasms, will not find satisfaction here.

Software is written within a particular development culture, by people having their own unique and changeable behavior patterns. Measurements of the products and processes in this environment are intrinsically noisy and are likely to include variables of influence that are not measured. This situation does not mean that analysis of the available measurements is a futile activity, what it means is that the uncertainty and variability is likely to be much larger than typically found in other engineering disciplines.

Statistics does not assign a meaning to the patterns it uncovers; interpreting the patterns thrown up by statistical analysis, to give them meaning, is your job dear reader (based on your knowledge and experience of the problem domain).

The tool used for statistical analysis is the R system. R was chosen because of its extensive ecosystem; there are many books, covering a wide range of subject areas, using R and active online forums discussing R usage (answers to problems can often be found by searching the Internet or if none are found a question can be posted with a reasonable likelihood of receiving an answer).

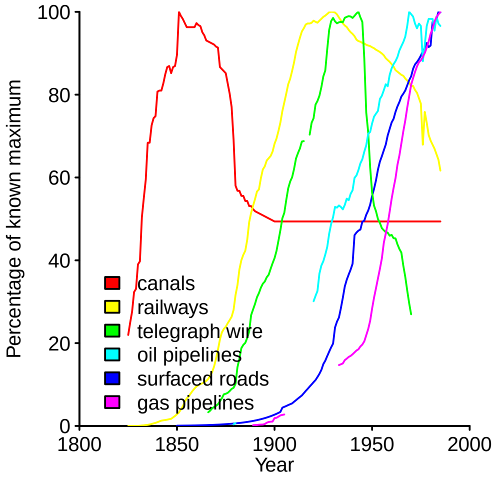


Figure 1.4: Growth of transport and product distribution infrastructure in the USA (underlying data is measured in miles). Data from Grübler et al.[741] Github–Local

The data and R code used in this book are freely available for download from the book's website.[927]

Like software development, data analysis contains a craft component, and the way to improve craft skills is to practice.

In January 2018 the U.S. Congress passed the Foundations for Evidence-Based Policymaking Act of 2018,[386] whose requirements include: "(1) A list of policy-relevant questions for which the agency intends to develop evidence to support policymaking. (2) A list of data the agency intends to collect, use, or acquire to facilitate the use of evidence in policymaking."

1.1 Software markets

The last 50 years, or so, has been a sellers market; the benefits provided by software systems has been so large that companies that failed to use them risked being eclipsed by competitors. Whole industries have been engulfed, and companies saddled with another Red Queen,[129] loosing their current position if they failed to keep running.

Provided software development projects looked like they would deliver something that was good enough, those involved knew that the customer would wait and pay; complaints received a token response. Software vendors learned that one way to survive in a rapidly evolving market was to get products to market quickly, before things moved on.

Experience gained from the introduction of earlier high-tech industries suggests that it takes many decades for major new technologies to settle down and reach market saturation.[1449] For instance, the transition from wood to steel for building battleships,[1415] started in 1858 and reached it zenith during the second world war; there is a long history of growth and decline of various forms of infrastructure (see figure 1.4). Various models of the evolution of technological innovations have been proposed.[1605]

Over the last 70 years a succession of companies have dominated the computing industry. The continual reduction in the cost of computing platforms created new markets, and occasionally one of these grew to become the largest market, financially, for computing resources. A company dominates the computer industry when it dominates the market that dominates the industry. Once dominant companies often continue to grow within their market, but their market is no longer the largest market for computers.

Figure 1.5 illustrates the impact of the growth of new markets on the market capitalization of three companies; IBM dominated when mainframes dominated the computer industry, the desktop market grew to dominate the computer industry and Microsoft dominated, smartphones removed the need for computers sitting on desks, and these have grown to dominate the computer market with Apple being the largest company in this market (Google's Android investment was a defensive move to ensure they are not locked out of advertising on mobile, i.e., it is not as intended to be a direct source of revenue, making it extremely difficult to estimate its contribution to Google's market capitalization). Figure 1.5 shows market capitalization (upper), and as a percentage of the top 100 listed US tech companies (lower), as of the first quarter of 2015.[520]

The three major eras, each with its own particular product and customer characteristics, have been (figure 1.6 shows sales of major computing platforms):

- the IBM era (sometimes known as the *mainframe* era, after the dominant computer hardware, rather than the dominant vendor): The focus was on business customers, with high priced computers sold, or rented, to large organizations who either rented software from the hardware vendor or paid for software to be developed specifically for their own needs. Large companies were already spending huge sums employing the (tens of) thousands of clerks needed to process the paperwork used in the control their businesses;[170] large companies could make large savings, in time and money, by investing in bespoke software systems, paying for any subsequent maintenance, and the cost of any faults experienced.

  When the actual cost of software faults experienced by one organization is very high (with potential for even greater costs if things go wrong), and the same organization is paying all, or most of the cost of create the software, that organization can see a problem that is costing it money, that it thinks it should have control over. Very large organizations are in a position to influence research agendas to target the problems they want solved.

  Large organizations tend to move slowly. The rate of change was slow enough for experience and knowledge of software engineering to be considered essential to do the job (this is not to say that anybody had to show that their skill was above some a minimum level before they would be employed as a software developer).

- the Wintel era: the Personal Computer running Microsoft Windows, using Intel's x86 processor family, was the dominant computing platform. The dramatic reduction in the cost of owning a computer significantly increases the size of the market, and it becomes commercially profitable for companies to create and sell software packages. The direct cost of software maintenance is not visible to these customers, but they pay the costs of the consequences of faults they experience when using the software.

  Microsoft's mantra of a PC on every desk required that people write software to cover niche markets. The idea that anyone could create an application was promoted as a means of spreading Windows, by encouraging people with application domain knowledge to create software running under MS-DOS and later Windows.

  Programming languages, libraries and user interface experienced high rates of change, which meant that developers found themselves on a relearning treadmill. An investment in learning had short payback periods; becoming an expert was not cost effective.

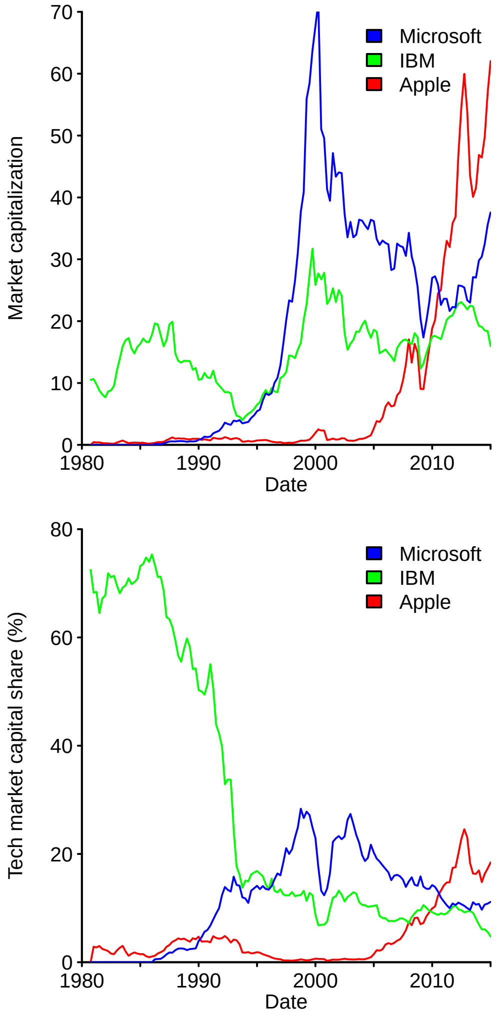


Figure 1.5: Market capitalization of IBM, Microsoft and Apple (upper), and expressed as a percentage of the top 100 listed US tech companies (lower). Data extracted from the Economist website.[520] Github–Local

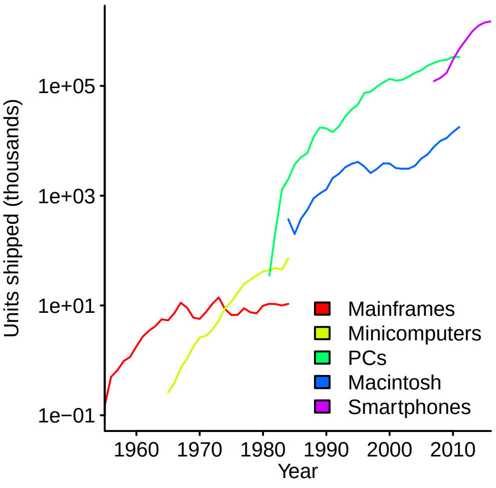


Figure 1.6: Total annual sales of some of the major species of computers over the last 60 years. Data from Gordon[707] (mainframes and minicomputers), Reimer[1551] (PCs) and Gartner[648] (smartphones). Github–Local

- the Internet era, no single vendor dominates the Internet, but some large niches have dominant vendors, e.g., mobile phones,

  It is difficult to judge whether the rate of change has been faster than in previous eras, or the volume of discussion about the changes has been higher because the changes have been visible to more people, or the lack of a dominant vendor to prevent change occurring too quickly.

  Mobile communications is not the first technology to set in motion widespread structural changes to industrial ecosystems. Prior to electrical power becoming available, mechanical power was supplied by water and then steam. Mechanical power is transmitted in straight lines using potentially dangerous moving parts; factories had to be organized around the requirements of mechanical power transmission. Electrical power transmission does not suffer from the restrictions of mechanical power transmission. It took some time for the benefits of electrical power (e.g., machinery did not have to be close to the generator) to diffuse through industry.[432]

The ongoing history of new software systems and computing platforms has created an environment where people are willing to invest their time and energy creating what they believe will be the next big thing. Those with the time and energy to do this, are often the young and inexperienced, outsiders in the sense that they don't have any implementation experience with existing systems. If any of these new systems take off, the developers involved will have made, or will make, many of the same mistakes made by the developers involved in earlier systems. The rate of decline of major software platforms is slow enough that employees with significant accumulated experience and expertise can continue to enjoy their seniority in well-paid jobs and have no incentive to jump ship to apply their expertise to an emerging system.

Mobile computing is only commercially feasible when the cost of computation, measured in Watts of electrical power, can be supplied by user-friendly portable batteries. Figure 1.7 shows the decline in electrical power consumed by a computation between 1946 and 2009; historically, it has been halving every 1.6 years.

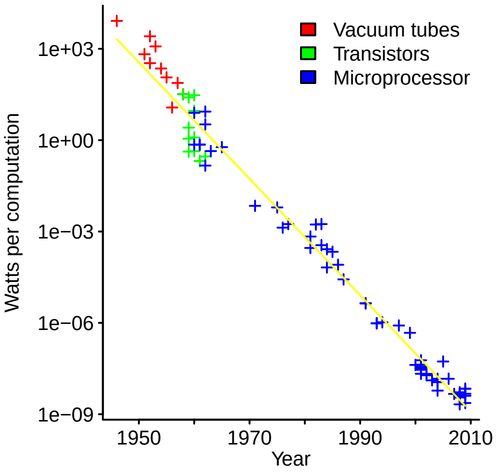


Figure 1.7: Power consumed, in Watts, executing an instruction on a computer available in a given year. Data from Koomey et al.[1023] Github–Local

Software systems have yet to reach a stable market equilibrium in many of the ecosystems they have colonised. Many software systems are still new enough that they are expected to adapt when the ecosystem in which they operate evolves. The operational procedures of these systems have not yet been sufficiently absorbed into the fabric of life that they enjoy the influence derived from users understanding that the easiest and cheapest option is to change the world to operates around them.

Economic activity is shifting towards being based around intangible goods;[775] cognitive capitalism is becoming mainstream.

A study by Goodridge, Haskel and Wallis[699] estimated the UK investment in intangible assets, as listed in the audited accounts that UK companies are required to file every year. Figure 1.8 shows the total tangible (e.g., buildings, machinery and computer hardware) and intangible assets between 1990 and 2012. Economic competencies are items such as training and branding, Innovative property includes scientific R&D, design and artistic originals (e.g., films, music and books); accounting issues associated with software development are discussed in chapter 3.

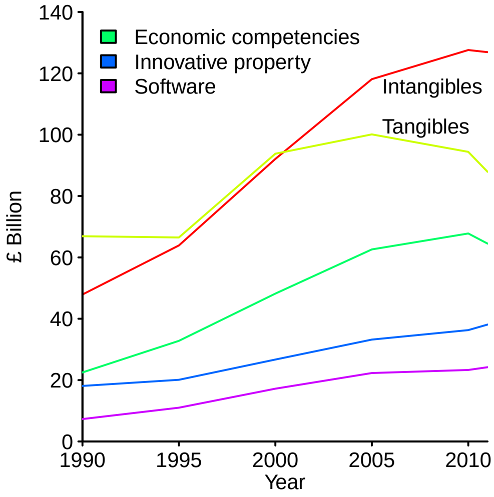


Figure 1.8: Total investment in tangible and intangible assets by UK companies, based on their audited accounts. Data from Goodridge et al.[699] Github–Local

Some companies treat some software as fixed-assets. Figure 1.9 shows quarterly totals for newly purchased and own software, and computer hardware, reported by UK companies as new fixed-assets.

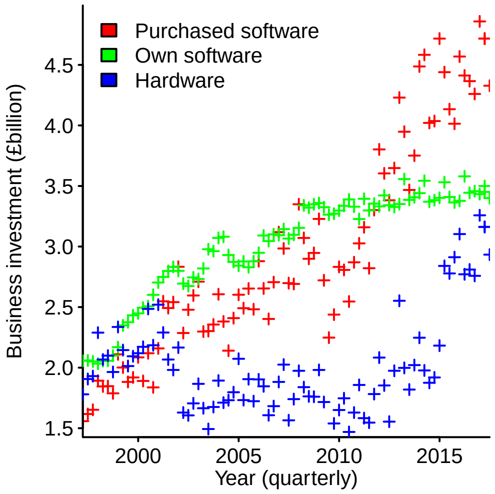


Figure 1.9: Quarterly value of newly purchased and own software, and purchased hardware, reported by UK companies as fixed-assets. Data from UK Office for National Statistics.[1385] Github–Local

A study by Wang[1900] found that while firms associated with the current IT fashion have a higher reputation and pay their executives more, they do not have higher performance. As companies selling hardware have discovered, designing products to become technologically obsolete (perceived or otherwise),[1700] or wear out and cease to work after a few years,[1927] creates a steady stream of sales. Given that software does not wear out, the desire to not be seen as out of fashion provides a means for incentivizing users to update to the latest version.

Based on volume of semiconductor sales (see figure 1.10), large new computer-based ecosystems are being created in Asia; the rest of the world appears to have reached the end of their growth phase.

The economics of chip fabrication,[1042] from which Moore's law derived,[iii] was what made the firestorm possible; the ability to create more products (by shrinking the size of components; see figure 1.11) for roughly the same production cost (i.e., going through the chip

[iii] Moore's original paper,[1292] published in 1965, extrapolated four data-points to 1975 and questioned whether it would be technically possible to continue the trend

fabrication process);[837] faster processors and increased storage capacity were fortuitous, customer visible, side effects. The upward ramp of the logistic equation has now levelled off,[603] and today Moore's law is part of a history that will soon be a distant memory.

Software systems are part of a world-wide economic system that has been rapidly evolving for several hundred years; the factors driving economic growth are slowly starting to be understood.[913] Analysis of changes in World GDP have found cycles, or waves, of economic activity; Kondratieff waves are the longest, with a period of around 50 years (data from more centuries may contain a longer cycle), a variety of shorter cycles are also seen, such as the Kuznets swing of around 20 years. Five Kondratieff waves have been identified with major world economic cycles, e.g., from the industrial revolution to information technology.[1449] Figure 1.12 shows a spectral analysis of estimated World GDP between 1870 and 2008; adjusting for the two World-wars produces a smoother result.

While the rate at which new technologies have spread to different countries has been increasing over time,[379] there has still been some lag in the diffusion of software systems[397] around the world.

Computers were the enablers of the latest wave, from the electronic century.

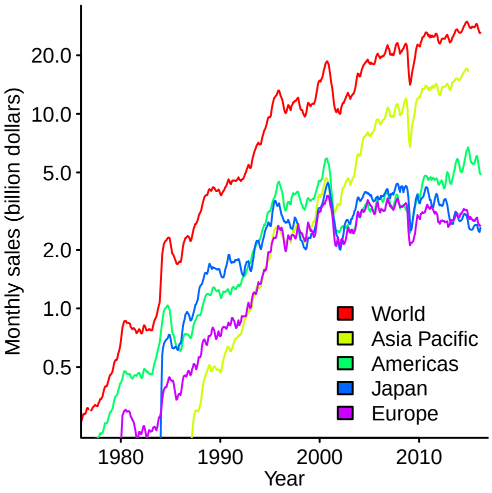


Figure 1.10: Billions of dollars of worldwide semiconductor sales per month. Data from World Semiconductor Trade Statistics.[1959] Github–Local

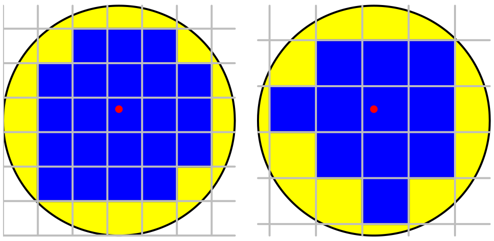


Figure 1.11: Smaller component size allows more devices to be fabricated on the same slice of silicon, plus material defects impact a smaller percentage of devices. Github–Local

1.1.1 The primary activities of software engineering

Software engineering is the collection of activities performed by those directly involved in the production and maintenance of software.

These activities have some path dependency: once the know-how and infrastructure for performing some activity becomes widely used, this existing practice is likely to continue to be used.

Perhaps the most entrenched path dependency in software development is the use of two-valued logic, i.e., binary. The most efficient radix, in terms of representation space (i.e., number of digits times number of possible values of each digit), is: 2.718...,[786] whose closest integral value is 3. The use of binary, rather than ternary, has been driven by the characteristics of available electronic switching devices.[iv] Given the vast quantity of software making an implicit, and in some cases explicit, assumption that binary representation is used, a future switching technology that would support the use of a ternary representation might not be adopted or be limited to resource constrained environments.[1909]

Traditionally, software development activities have included: obtaining requirements, creating specifications, design at all levels, writing and maintaining code, writing manuals fixing problems and providing user support. Large organizations compartmentalise activities and the tasks assigned to software developers tend to be purely software related.

In small companies there is greater opportunity, and sometimes a need, for employees to become involved in tasks that would not be considered part of the job of a software developer in a larger company. For instance, being involved in any or all of a company's activities from the initial sales inquiry through to customer support of the delivered system; the financial aspect of running a business is likely to be much more visible in a small company.

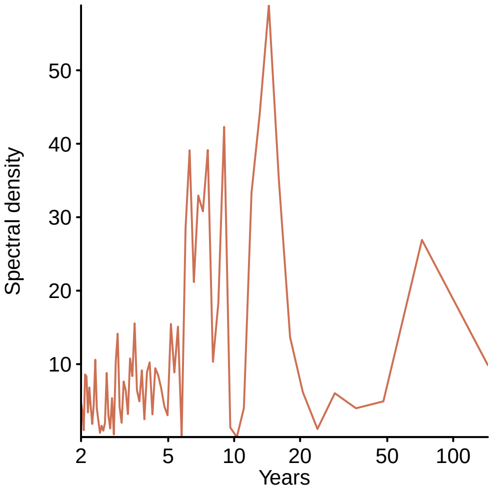


Figure 1.12: Spectral analysis of World GDP between 1870-2008; peaks around 17 and 70 years. Data from Maddison.[1176] Github–Local

Some software development activities share many of the basic principles of activities that predate computers. For instance, user interface design shares many of the characteristics of stage magic.[1817]

Software is created and used within a variety of ecosystems, and software engineering activities can only be understood in the context of the ecosystem in which it operates.

While the definition of software engineering given here is an overly broad one, let's be ambitious and run with it, allowing the constraints of data availability and completing a book to provide the filtering.

The debate over the identity of computing as an academic discipline is ongoing.[1798]

[iv] In a transistor switch, Off is represented by very low-voltage/high-current and On represented by saturated high-voltage/very low-current. Transistors in these two states consume very little power (power equals voltage times current). A third state would have to be represented at a voltage/current point that would consume significantly more power. Power consumption, or rather the heat generated by it, is a significant limiting factor in processors built using transistors.

1.2 History of software engineering research

The fact that software often contained many faults, and took much longer than expected to produce, was a surprise to those working at the start of electronic computing, after World War II. Established practices for measuring and documenting the performance of electronics were in place and ready to be used for computer hardware,[1015, 1463] but it was not until the end of the 1960s that a summary of major issues appeared in print.[1337]

Until the early 1980s most software systems were developed for large organizations, with over 50% of US government research funding for mathematics and computer science coming from the Department of Defense,[601] an organization that built large systems, with time-frames of many years. As customers of software systems, these organizations promoted a customer orientated research agenda, e.g., focusing on minimizing customer costs and risks, with no interest in vendor profitability and risk factors. Also, the customer is implicitly assumed to be a large organization.

Very large organizations, such as the DOD, spend so much on software it is cost effective for them to invest in research aimed at reducing software costs, and learning how to better control the development process. During the 1970s project data, funding and management by the Rome Air Development Center,[v] RADC, came together to produce the first collection of wide-ranging, evidence-based, reports analysing the factors involved in the development of large software systems.[464, 1806]

For whatever reason the data available at RADC was not widely distributed or generally known about; the only people making use of this data in the 1980s and 1990s appear to be Air Force officers writing Master's theses.[1410, 1678]

The legacy of this first 30 years was a research agenda oriented towards building large software systems.

Since around 1980, very little published software engineering research has been evidence-based (during this period, not including a theoretical contribution in empirical research was considered grounds for rejecting a paper submitted for publication[13]). In the early 1990s, a review[1157] of published papers relating to software engineering, found an almost total lack of evidence-based analysis of its engineering characteristics; a systematic review of 5,453 papers published between 1993 and 2002[768] found 2% reporting experiments. When experiments had been performed, they suffered from small sample sizes[956] (a review[1895] using papers from 2005 found that little had changed), had statistical power falling well below norms used in other disciplines[515] or simply failed to report an effect size (for the 92 controlled experiments published between 1993 and 2002 only 29% reported an effect size[956]).

Why have academics working in an engineering discipline not followed an evidence-based approach in their research? The difficulty of obtaining realistic data is sometimes cited[1506] as the reason; however, researchers in business schools have been able to obtain production data from commercial companies,[vi] perhaps the real reason is those in computing departments are more interested in algorithms and mathematics, rather than the human factors and economic issues that dominate commercial software development. The publication and research culture within computing departments may have discouraged those wanting to do evidence-based work (a few intrepid souls did run experiments using professional developers[136]). Researchers with a talent for software engineering either moving on to other research areas or to working in industry, leaving the field to those with talents in the less employable areas of mathematical theory, literary criticism (of source code) or folklore.[260]

When statistical analysis has been used, the techniques applied[448] have often been a hold-over from pre-computer times, when calculations had to be done by hand, i.e., techniques that could be performed manually, sometimes of low power and requiring the data to have specific characteristics. Also, the methods used to sample populations have been not been rigorous.[124]

Human psychology and sociology continue to be completely ignored as major topics of software research, a fact pointed out over 35 years ago.[1664] The irrelevance of most existing software engineering research to industry is itself the subject of academic papers.[647]

[v] The main US Air Force research lab. There is probably more software engineering data to be found in US Air Force officers' Master's theses, than all academic software engineering papers published before the early 2000s.

[vi] Many commercial companies do not systematically measure themselves and maintain records, so finding companies that have data requires contacts and persistence.

A lack of evidence has not prevented researchers expounding plausible sounding theories that, in some cases, have become widely regarded as true. For instance, it was once claimed, without any evidence, that the use of source code clones (i.e., copying code from another part of the project) is bad practice (e.g., clones are likely to be a source of faults, perhaps because only one of the copies was updated).[620] In practice, research has shown that the opposite is true,[1534, 1814] clones are less likely to contain faults than *uncloned* source.

Many beliefs relating to software engineering processes, commonly encountered in academic publications, are based on ideas created many years ago by researchers who were able to gain access to a relevant (often tiny) dataset. For instance, Perry[1451] divided software interface faults into 15 categories using a data set of just 85 modification requests to draw conclusions; this work is still being cited in papers 25 years later. These fossil theories have continued to exist because of the sparsity of data needed to refute or improve on them.

It is inevitable that some published papers contain claims about software engineering that turn out to be roughly correct; coincidences happen. Software engineering papers can be searched for wording which can be interpreted as foreseeing a recent discovery, in the same way it is possible to search the prophecies of Nostradamus to find one which can be interpreted as predicting the same discovery.

The quantity of published papers on a topic should not be confused with progress towards effective models of behavior. For instance, one study[1041] of research into software process improvement, over the last 25 years, found 635 papers, with experience reports and proposed solutions making up two-thirds of publications. However, proposed solutions were barely evaluated, there were no studies evaluating advantages and disadvantages of proposals, and the few testable theories are waiting to be tested.

Over the last 10 years or so, there has been an explosion of evidence-based research, driven by the availability of large amounts of data extracted from open source software. However, existing theories and ideas will not change overnight. Simply ignoring all research published before 2005 (roughly when the public data deluge started) does not help, earlier research has seeded old wives tales that have become embedded in the folklore of software engineering creating a legacy that is likely to be with us for sometime to come.

A study by Rousseau, Di Cosmo and Zacchiroli[1589] tracked the provenance of source code in the Software Heritage archive. Figure 1.13 shows the number of unique blobs (i.e., raw files) and commits that first appeared in a given month, based on date reported by file system or version control (the doubling time for files is around 32 months, and for commits around 23 months). Occurrences at future dates put a lower bound on measurement noise.

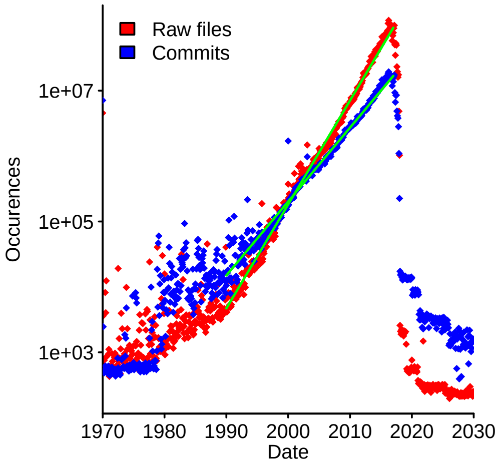


Figure 1.13: Number of unique files and commits first appearing in a given month; lines are fitted regression models of the form: $Files \propto e^{0.03 months}$ and $Commits \propto e^{0.022 months}$. Data kindly provided by Rousseau.[1589] Github–Local

This book takes the approach that, at the at the time of writing, evidence-based software engineering is essentially a blank slate. Knowing that certain patterns of behavior regularly occur is an empirical observation; a theory would make verifiable predictions that include the observed patterns. In some cases existing old wives tales are discussed, when it is felt that their use in an engineering environment would be seriously counter-productive. For instance, while various software metrics (e.g., Halstead's metric) are widely known, your authors' experience is that practicing developers do not invest effort in using them; they are famous for being famous, and so there is little to be gained in spending much effort debunking them (which can be counter-productive[1105]).

1.2.1 Folklore

The dearth of experimental evidence has left a vacuum that has been filled by folklore and old-wives' tales. Examples of software folklore include claims of a 28-to-1 productivity difference between best/worst developers, and that parameter passing in function calls is resource intensive for embedded systems.

The productivity claim, sometimes known as the *Grant-Sackman study*, is based on a casual reading of an easily misinterpreted table appearing in a 1968 paper[1602] by these authors. The paper summarised a study that set out to measure the extent to which the then new time-sharing approach to computer usage, was more productive for software development than existing batch processing systems (where jobs were typically submitted via punched cards, with programs executing (sometimes) hours later, followed by their output printed on paper); the table listed the ratio between the best subject performance using the fastest system, and the worst subject performance on the slowest system (the 28:1 ratio included programmer and systems performance differences, and if batch/time-sharing differences are separated out the maximum difference ratio is 14:1, the minimum 6:1). The actual measurement data was published[719] in a low circulation journal, and was immediately followed by a strongly worded critique;[1062] see fig 8.22.

A 1981 study by Dickey[487] separated out subject performance from other factors, adjusted for individual subject differences, and found a performance difference ratio of 1:5. However, by this time the 28:1 ratio had appeared in widely read magazine articles and books, and had become established as *fact*. In 1999, a study by Prechelt[1502] explained what had happened, for a new audience.

Embedded software runs on resource limited hardware, which is often mass-produced, and saving pennies per device can add up to a lot of money; systems are populated with the smallest possible memory and power consumption is reduced by using the slowest possible clock speeds, e.g., closer to 1 MHz than 1 GHz.

The Grant-Sackman study has not only been misinterpreted, it involves a development environment that has ceased to exist. Embedded systems development has a culture that is strongly intertwined with the hardware used, and hardware has been continually getting more powerful.

Experienced embedded developers are aware of hardware performance limitations they have to work within. Many low-cost processors have a very simple architecture with relatively few instructions and parameter passing, to a function, can be very expensive (in execution time and code size) compared to passing values to functions in global variables on some processors.

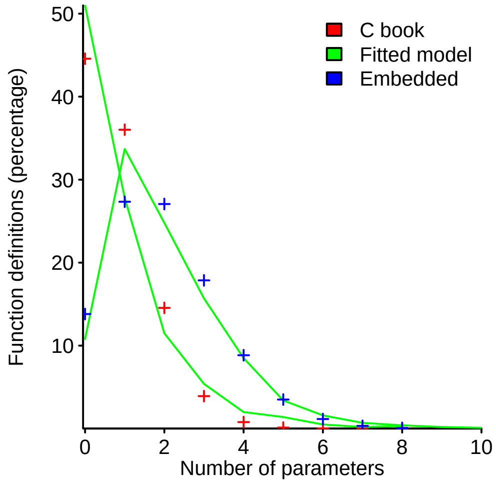


Figure 1.14: Percentage of function definitions declared to have a given number of parameters in: embedded applications, and the translated form of a sample of C source code. Data for embedded applications kindly supplied by Engblom,[537] C source code sample from Jones.[919] Github–Local

A study by Engblom[537] investigated differences in the characteristics of embedded C software, and the SPECint95 benchmark. Figure 1.14 shows the percentage of function definitions containing a given number of parameters, for embedded software the SPECint95 benchmark and desktop software measured by Jones.[919] A Poisson distribution provides a reasonable fit to both sets of data; for desktop software, the distribution of function definitions having a given number of parameters the Poisson distribution has $\lambda = 2$, while for embedded developers $\lambda = 0.8$.

These measurements were of source code from the late 1990s; have embedded systems processor characteristics changed since then?

Today, companies are likely to be just as interested in profit, e.g., saving pennies. Compilers may have become better at reducing function parameter overheads for some processor, but it is beliefs that drives developer usage.

Embedded devices have become more mainstream, with companies selling IoT devices with USB interfaces. This availability provides an opportunity for aspects of desktop and mobile system development culture to invade the culture of embedded development. In some cases, where code size or/and performance is critical, developers looking for savings may learn about the overheads of parameter passing. Within existing embedded system communities, past folklore may no longer apply (because the hardware has changed).

Is the value of λ always approximately 0.8 or 2.0? Is there a range of values, depending on developer experience (old habits die hard and parameter overhead will depend on processor characteristics, e.g., 4-bit, 8-bit and 16-bit processors)?

1.2.2 Research ecosystems

Interactions between people who build software systems and those involved in researching software has often suffered from a misunderstanding of each other's motivations and career pressures.

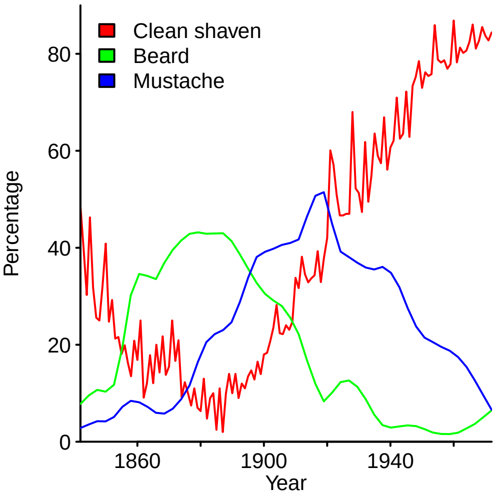


Figure 1.15: Changing habits in men's facial hair. Data from Robinson.[1571] Github–Local

Until the 1980s a significant amount of the R&D behind high-tech products was done in commercial research labs (at least in the US[73]). For instance, the development of Unix and many of its support tools (later academic derived versions were reimplementations of the commercial research ideas). Since then many commercial research labs have been shut or spun-off, making universities the major employer of researchers in some areas.

Few people in today's commercial world have much interaction with those working within the ecosystems that are claimed to be researching their field. The quaint image of researchers toiling away for years before publishing a carefully crafted manuscript is long gone.[524] Although academics continue to work in a feudal based system of patronage and reputation, they are incentivised by the motto “publish or perish”,[1433] with science perhaps advancing one funeral at a time.[94] Hopefully the migration to evidence-based software engineering research will progress faster than fashions in men’s facial hair (most academics researching software engineering are men); see figure 1.15.

Academic research projects share many of the characteristics of commercial start-ups. They involve a few people attempting to solve a sometimes fuzzily defined problem, trying to make an improvement in one area of an existing *product*, and they often fail, with the few that succeed producing spectacular returns. Researchers are serial entrepreneurs in that they tend to only work on funded projects, moving onto other projects when funding runs out (and often having little interest in previous projects). Like commercial product development, the choice of research topics is fashion driven; see figure 1.16.

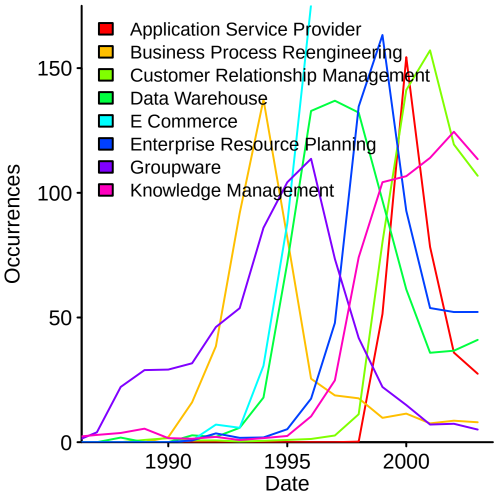


Figure 1.16: Number of papers, in each year between 1987 and 2003, associated with a particular IT topic. The E-commerce paper count peaks at 1,775 in 2000 and in 2003 is still off the scale compared to other topics. Data kindly provided by Wang.[1900] Github–Local

The visible output from academic research are papers published in journals and conference proceedings. It is important to remember that: " . . . an article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment, and the complete set of instructions, which generated the figures."[266]

Many journals and conferences use a process known as *peer review* to decide which submitted papers to accept. The peer review process[549] involves sending submitted papers to a few referees, ideally chosen for their expertise of the topic covered, who make suggestions on how to improve the paper and provide a yes/no/maybe acceptance decision (the identity of the paper’s author(s), and the referees are often anonymous). The peer review process first appeared in 1665, however, it only became widely used in the 1970s in response to concerns over public accountability of government funded research.[121] It now looks to be in need of a major overhaul[1504] to improve turn-around time, and handle the work load generated by a proliferation of journals and conferences, as well as addressing corruption;[1961] some journals and conferences have become known for the low quality of their peer reviews, even accepting fake papers.[1051] The status attached to the peer review process has resulted in it being adopted by journals covering non-traditional fields, e.g., psychic research.

In the past most researchers have made little, if any, effort to archive the data they gather for future use. One study[1884] requested the datasets relating to 516 biology related studies, with ages from 2 to 22 years; they found that the probability of the dataset being available fell by 17% per year. Your author’s experience of requesting data from researchers is that it often fails to survive beyond the lifetime of the computer on which it was originally held.

Researchers may have reasons, other than carelessness, for not making their data generally available. For instance, a study[1932] of 49 papers in major psychology journals found that the weaker the evidence for the researcher’s hypothesis the less likely they were to be willing to share their data.

The importance of making code and data available is becoming widely acknowledged, with a growing number of individual researchers making code/data available for download, and journals having explicit policies about authors making code/data available.[1761] In the UK researchers who receive any funding from a government research body are now required to archive their data, and make it generally available.[1555]

For some time now, academic performance has often been measured by number of papers published, and the impact factor of the journal in which they were published[1045] (scientific journals publish a percentage of the papers submitted to them, with prestigious high impact journals publishing a lower percentage than those have lower impact factors; journals and clubs share a common economic model[1496]). Organizations that award grants to researchers often consider the number of published papers and impact factor of the publication journal, when deciding whether to fund a grant application; the effect is to generate an evolutionary pressure that selects for bad science[1708] (as well as predatory journals[871] that accept any submitted paper, provided the appropriate fee is paid).

One consequence of the important role of published paper count in an academic’s career, is an increase in scientific fraud,[563] most of which goes undetected;[374] one study[565] found that most retracted papers (67.4%) were attributable to misconduct, around a 10-fold increase since 1975. More highly ranked journals have been found to publish a higher percentage of retracted papers,[246] it is not known whether this represents an increased in flawed articles, or an increase in detection.[1748] Only a handful of software engineering papers have been retracted, perhaps a lack of data makes it very difficult to verify the claims made by the authors. The website retractionwatch.com reports on possible and actual paper retractions.

Figure 1.17 shows the number of citations, in this book, to research published in a given year, plus the number of associated datasets; lines are fitted regression models.

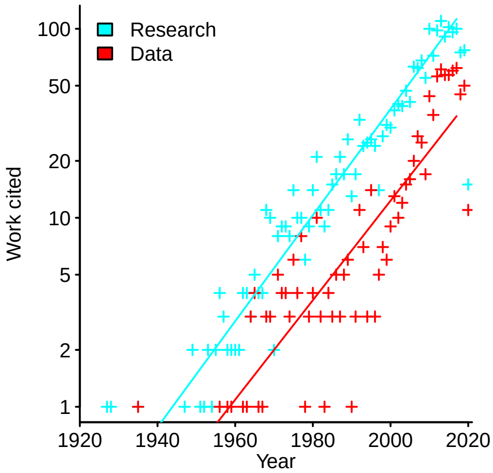


Figure 1.17: Number of articles appearing in a given year, cited in this book, plus number of corresponding datasets per year; both fitted regression lines have the form: $Citations \propto e^{0.06Year}$. Github–Local

Pointing out that academics are often more interested in career development than scientific development, and engage in questionable research practices is not new; Babbage complained about this behavior in 1830.[96]

Commercial research labs Many large companies have research groups. While the researchers working in these groups often attend the same conferences as academics and publish papers in the same journals; their performance is often measured by the number of patents granted.

Citation practices This book attempts to provide citations to any factual statements, so readers can perform background checks. To be cited papers have to be freely available for public download, unless published before 2000 (or so), and when data is analysed it has to be free for public distribution.[vii] Programs, packages and libraries are not cited, while sometimes others do cite software.[1112]

When academics claim their papers can be freely downloaded, what they may mean is that the university that employs them has paid for a site-wide subscription that enables university employees to download copies of papers published by various commercial publishers. Taxpayers pay for the research and university subscriptions, and most of these taxpayers do not have free access to it. Things are slowly changing. In the UK researchers in receipt of government funding are now incentivized to publish in journals that will make papers produced by this research freely available after 6-12 months from publication date. Your author's attitude is that academics are funded by taxpayers, and if they are unwilling to provide a freely downloadable copy on their web page, they should not receive any credit for the work.[viii]

Software developers are accustomed to treating documents that are more than a few years old, as being out-of-date; a consequence of the fast changing environment in which they often operate. Some fields, such as cognitive psychology, are more established, and people do not feel the need to keep repeating work that was first performed decades ago (although it may be replicated as part of the process of testing new hypothesis). In more established fields, it is counter-productive to treat date of publication as a worthiness indicator.

Researchers being disconnected from the practical realities of their field is not unique to software engineering; other examples include medicine[1396] and high-energy physics.[849]

1.2.3 Replication

The gold standard of the scientific method is the controlled randomised experiment, followed by replication of the results by others (findings from earlier studies failing to replicate is common[882]). In this kind of experiment, all the factors that could influence the outcome of an experiment are either controlled or randomly divided up such that any unknown effects add noise to the signal rather than spurious ghost patterns.

One study[1501] of medical practices, based on papers published between 2001 and 2010 addressing a medical practice, found that 40% of follow-up studies reversed the existing practice and 48% confirmed it.

Discovering an interesting pattern in experimental data is the start, not the end of experimentation involving that pattern (the likelihood of obtaining a positive result is as much to do with the subject studied as the question being asked[564]). The more often behavior is experimentally observed the greater the belief that the effect seen actually exists. Replication is the process of duplicating the experiment(s) described in a report or paper to confirm the findings previously obtained. For replication to be practical, researchers need to be willing to share their code and data, something that a growing numbers of software engineering researchers are starting to do; those attempting replication are meeting with mixed success.[378]

[vii] The usual academic procedure is to cite the first paper that proposes a new theory, describes a pattern of behavior, etc.

[viii] In fact they should not have been funded in the first place; if an academic refuses to make a copy of their papers freely available to you, please report their behavior to your elected representative.

Replication is not a high status activity, with high impact journals interested in publishing research that reports new experimental findings, and not wanting to dilute their high impact status by publishing replications (which, when they are performed and fail to replicate previous results considered to be important discoveries, can often only find a home for publication in a low impact journal). A study[1399] replicating 100 psychology experiments found that while 97% of the original papers reported p-values less than 0.05, only 36% of the replicated experiments obtained p-values this low; a replication study[315] of 67 economics papers was able to replicate 22 (33%) without assistance from the authors, 29 with assistance.

Replication is necessary,[175] i.e., the results claimed by one researcher need to be reproducible by other people, if they are to be accepted. Without replication researchers are amassing a graveyard of undead theories.[587]

1.3 Applying the findings

Your author set out to analyze all the publicly available software engineering data. As of early 2020, ?650? datasets have been collected and analyzed.

How might the findings from the analysis of these ?650? datasets be applied in practice? Perhaps the two main applications of the behavior patterns highlighted by the analysis are:

- helping to rule out behaviors that might otherwise be thought possible. In situations where the outcome is uncertain, being able to rule out a range of possibilities reduces costs by decreasing investment in possibilities that are unlikely to occur,
- provide ideas about the behaviors of the processes producing the behaviors found. Nothing more specific can be inferred because most of the analyses are of data obtained from one experiment, measurements of a collection of projects at one point in time or a few projects over an extended period.

  Many plots include one or more lines showing fitted regression models. The purpose of fitting these models is to highlight patterns that may be present. Most of these models were created by your author after seeing the data, what is sometimes known as *HARKing* (Hypothesizing After the Results are Known),[978] or less technically they are just-so stories created from a fishing expedition. This is not how rigorous science is done.

Software development involves many interacting processes. Building a reasonably accurate models of these processes is going to need substantially more data than is analyzed in this book. These models will be created by weaving together results from replicated experiments.

Figure 1.17 shows that evidence-based research is rapidly growing. Researchers also need to move towards studies whose results can be woven with results from other studies to be part of a larger whole, rather than standing-alone.

Open Source projects are readily available in quantity, and a lot of empirical research now makes use of this public resource. To what extent are findings from the analysis of Open Source projects applicable to non-Open Source software development (researchers often use some popularity metric, such as Github stars, to filter out projects that have not attracted enough attention from others)? The definitive answer can only come from comparing findings from software systems developed in both environments (enterprise-driven open source is available[1728]).

There are threats to the validity of findings based on Open Source packages, when applied to other Open Source packages, these include:

- the choice of Open Source packages to analyse is a convenience sample:
  - a variety of tools have been created for extracting data, with many tools targeting a particular language; mostly Java, with C being the second most popular,
  - using an easily accessible corpus of packages. Creating a corpus is a popular activity because it currently provides a low-cost route to a published paper for anyone with programming skills, and can potentially acquire many citations, via the researchers who use it,
- characteristics of Open Source development that generate large measurement biases are still being discovered. Some major sources of measurement bias, that have been found, include:

  - the timing of commits and identity of the committer may not be those of the developer who wrote the code. For instance, updates to the Linux kernel are submitted by developers to the person responsible for the appropriate subsystem, who forwards those that are accepted to Linus Torvalds, who integrates those that he accepts into mainline Linux; a lot of work is needed to extract an accurate timeline[662] from the hundreds of separate developer repositories,
  - a large percentage of fault reports have been found to be requests for enhancement.[486, 811]

The freely downloadable applications found in App stores are another popular research topic.

1.4 Overview of contents

This book contains two distinct halves:

- the first half discusses the major areas of software engineering, driven by an analysis of the publicly available data. The aim is to provide the information needed to reduce the resources needed to build and maintain software systems, and to make efficient use of available resources.

  Many topics usually covered in software engineering textbooks are not discussed because public data relating to them could not be located,
- the second half discusses the data analysis techniques applicable to the kinds of measurement data, and problems, likely to be encountered by software developers. The intended readership is software developers wanting to apply techniques, who don't want to learn about the mathematics that underlies their implementation.

  The results of the analysis are intended to help managers and developers understand the processes that generated the data.

Human cognition Human brains supply the cognitive effort that directs and performs the activities involved in software production. An understanding of the operating characteristics of the human brain is needed to maximise the effective cognitive effort delivered by those involved on a project. Characteristics such as cognitive capacity, memory storage/recall performance, learning, personality (e.g., propensity to take risks), and processing of visual information are discussed.

Software developers come preloaded with overlearned behaviors, derived from the native culture of their formative years, and at least one human language, which they have used for many hours when reading.

Cognitive capitalism Economic issues, as they relate to software, are discussed. Current economic theories and practices are predominantly based around the use of labor as the means of production; the economics of the products of intellectual effort has been growing in importance for some time (the chapter title provides a direction for how to think about the economic material).

The approach to software economics is from the perspective of the software engineer or software company, rather than the perspective of the customer or user of software (which differs from much existing work, which adopts the customer or user perspective).

Ecosystems Software is created in a development ecosystem, and is used in a customer ecosystem. Customer demand motivates the supply of energy that drives software ecosystems. Software does not wear out, and the motivation for change comes from the evolution of these ecosystems.

The rate of ecosystem evolution puts an upper limit on the productive lifetime of software. Lifetime is a key component of return on investment calculations.

Various developer ecosystems (e.g., careers), and development (e.g., APIs) ecosystems are discussed, along with non-software issues having an impact on developers.

Projects Incentive structures are a key component to understanding the creation of software systems, from bidding to delivery, and ongoing maintenance; risks and returns. Client/vendor interaction, and what's in it for developers and managers.

What are the factors affecting resource estimation, and how much accuracy is good enough? The phases and items of work occurring along the path to delivery are discussed.

Reliability Software systems have a minimum viable reliability, i.e., what will the market tolerate. The little that is known about fault experiences is discussed (the result of an interaction between input values and a coding mistakes); where mistakes occur, their lifetime, predicting population size, and the cost-effectiveness of reducing or removing mistakes.

Source code This is studied to find ways of reducing the resources needed to create and maintain it, resources such as the developer cognitive effort needed to process code.

What are desirable source code characteristics, and what can be learned from existing patterns of usage, i.e., where are the opportunities for investment in the production of source code?

Stories told by data There is no point analysing data unless the results can be effectively communicated to the intended audience. A compendium of examples of techniques that might be used to communicate a story found in data, along with issues to look out for in the stories told by others.

Statistics Developers are casual users of statistics and don't want to spend time learning lots of mathematics; they want to make use of techniques, not implement them. The approach used is similar to that of a Doctor examining a patient for symptoms that might suggest underlying processes.

It is assumed that readers have basic algebra skills, and can interpret graphs; no other statistical knowledge is assumed.

In many practical situations, the most useful expertise to have is knowledge of the application domain that generated the data, along with how any findings might be applied. It is better to calculate an approximate answer to the correct problem, than an exact answer to the wrong problem.

Because evidence-based software engineering has only recently started to be applied, there is uncertainty about which statistical techniques are most likely to be generally applicable. Therefore, an all encompassing approach is taken, and a broad spectrum of topics is covered, including:

- probability: making inferences about individual events based on the characteristics of the population (statistics makes inferences about the population based on the characteristics of a sample of the population). Concepts discussed include: probability distributions, mean and variance, Markov chains, and rules of thumb,
- statistics for software engineering: statistics could be defined as the study of algorithms for data analysis; algorithms covered include: sampling, describing data, p-value, confidence intervals, effect size, statistical power, bootstrapping, model building, comparing two or more groups,
- regression modeling: the hammer used to analyse much of the data in this book. The kind of equation fitted to data can provide information about the underlying processes that generated the data, and which variables have the most power to explain the behavior measured.

  Software engineering measurements come in a wide variety of forms, and while ordinary least-squares might be widely used in the social sciences, it is not always suitable for modeling software datasets; more powerful techniques such as generalized least-squares, nonlinear models, mixed models, additive models, structural equation models and others are discussed,
- time series: analysis of data where measurements at time t are correlated with measurements made earlier, e.g., at time $t-1$, is the domain of time series analysis (regression modeling assumes that successive measurements are independent of each other),
- survival analysis: measurements of time to an event occurring (e.g., death) are the domain of survival analysis,
- machine learning: various techniques for finding patterns in data, when the person doing the analysis has little or no knowledge of the data, or the application domain that produced it. Sometimes we are clueless button pushers, and machine learning can be a useful guide.

Experiments Probably the most common experiment performed by developers is benchmarking of hardware and software. The many difficulties and complications involved in performing reliable benchmarks are illustrated.

General issues involving the design of experiments are discussed.

Surveys: Analysis of data obtained by asking people questions.

Data cleaning Garbage in garbage out. Data cleaning is the penultimate chapter (going unmentioned in some books), and is often the most time-consuming part of data analysis and a very necessary activity for obtaining reliable results.

Common data cleaning tasks, along with possible techniques for detecting potential problems and solving them using R are discussed.

Overview of R An overview of R aimed at developers who are fluent in at least one other computer language. The discussion concentrates on those language features likely to be commonly used, but behave surprisingly differently from languages the reader is likely to be familiar with.

Obtaining and installing R: If you cannot figure out how to obtain and install R, this book is not for you.

RStudio is a widely used R IDE that is also sometimes used by your author.

1.4.1 Why use R?

The main reasons for selecting R as the language+support library in which to implement the statistical analysis programs used in this book are:

- it is possible to quickly write a short program that solves the kind of problems that often occur when analysing software engineering data. The process often follows the sequence: read data from one of a wide variety of sources, operate on it using functions selected from an extensive library of existing packages, and finally graphically display the results or print values,
- lots of data analysis people are using it: there is a very active ecosystem with many R books, active discussion forums where examples can be found, answers to common questions found and new questions posted,
- accuracy and reliability: a comparison of the reliability of 10 statistical software packages[1383] found that GAUSS, LIMDEP, Mathematica, MATLAB, R, SAS, and Stata provided consistent reliable estimation results, and a comparison of the statistical functions in Octave, Python, and R[39] found that R yielded the best results. A study[40] of the precision of five spreadsheets (Calc, Excel, Gnumeric, NeoOffice and Oleo), running under Windows Vista, Ubuntu, and macOS, found that no one spreadsheet provided consistently good results (it was recommended that none of these spreadsheets be used for nonlinear regression and/or Monte Carlo simulation); another study[1221] found significant errors in the accuracy of the statistical procedures in various versions of Microsoft Excel.
- an extensive library of add-on packages (over 15,000 at at the time of writing): CRAN, the Comprehensive R Archive Network is the official package library; some packages are only available on R-Forge and Github.

A freely available open source implementation is always nice to have.

1.5 Terminology, concepts and notation

Much of the basic terminology used in probability and statistics in common use today derives from gambling and experimental research in medicine and agriculture, because these were the domains where researchers working in the early days of statistics were employed.

- *group*: each sample is sometimes referred to as a group,
- *treatment*: The operation or process performed is often referred to as a treatment.
- *response variable* also known as a *dependent variable*, responds/depends to/on changes of values of explanatory variables; their behavior depends on the variables that are independent (at least of them),
- *explanatory variables* (also known as *independent*, *stimulus*, *predictor variables* is used when the variables are used to make predictions, *control variables* is sometimes used in an experimental setting), are used to explain, predict or stimulate the value of response variables; they are independent of the response variable,

- data is *truncated* when values below, or above some threshold are unobserved (or removed from the dataset).
- data is *censored* when values below, or above some threshold are set equal to the threshold,
- *between subjects*: when samples are obtained from different groups of subjects, often with the different groups performing a task under different experimental conditions,
- *within subjects*: Comparing two or more samples obtained using the same group of subjects, often with the different subjects performing a task under two or more different experimental conditions,
- a *parametric test* is a statistical technique that assumes the sample has some known distribution,
- a *nonparametric test* is a statistical technique that does not make any assumptions about the sample distribution (the term *distribution free test* is sometimes used).

The following are some commonly encountered symbols and notation:

- $n!$ (n factorial), denotes the expression $n(n-1)(n-2)\cdots 1$; calculated by R's `factorial` function,
- $\binom{n}{r} = \frac{n!}{r!(n-r)!}$, is a commonly occurring quantity in the analysis of probability problems; calculated by R's `choose` function,
- a hat above a variable, $\hat{y}$, denotes an estimate, in this case an estimate of y's value,
- μ (mu), commonly denotes the mean value; calculated by R's `mean` function,
- σ (sigma), commonly denotes the standard deviation; calculated by R's `sd` function. The terms 1-sigma, 2-sigma, etc. are sometimes used to refer to the probability of an event occurring. Figure 1.18 shows the sigma multiplier for various probabilities,
- $n \to \infty$ as n goes to infinity, i.e., becomes very very large,
- $n \to 0$, as n goes to zero, i.e., becomes very very small,
- $P(x)$, the probability of x occurring and sometimes used to denote the Poisson distribution with parameter x (however, this case is usually written using λ (lambda), e.g., $P(\lambda)$),
- $P(a < X)$, the probability that $a < X$. The functions `pnorm`, `pbinom` and `ppois` can be used to obtain the probability of encountering a value less than or equal to x for the respective distribution (e.g., Normal, Binomial and Poisson, as suggested by the naming convention),
- $P(|a-X|)$, the probability of the absolute value of the difference between a and X,
- $\prod_{i=1}^{6} a_i$, the product: $a_1 \times a_2 \times \cdots \times a_6$,
- $\sum_{i=1}^{6} P(a_i)$, the sum: $P(a_1) + P(a_2) + \cdots + P(a_6)$,

  The sum the probabilities of all the mutually exclusive things that could happen, when an action occurs, is always one. For instance, when a die is rolled, the six probabilities of a particular number occurring sum to one, irrespective of whether the die is fair, or has been tampered with in some way.
- $P(D|S)$, the probability of D occurring, given that S is true; known as the *conditional probability*. For instance, S might be the event that two dice have been rolled, and their face-up numbers sum to five, and D the event that the value of the first die is four.

  The value can be calculated as follows:

  $$P(D|S) = \frac{P(DS)}{P(S)}$$

  where $P(DS)$ is the probability of both D and S occurring together.

  If D and S are independent of each other, then we have: $P(DS) = P(D)P(S)$, and the above equation simplifies to: $P(D|S) = P(DS)$

  A lot of existing theory in statistics assumes that variables are independent. For instance, characteristics unique to each dice means that using a different dice for each roll will produce values having more independence than using the same dice for both rolls.

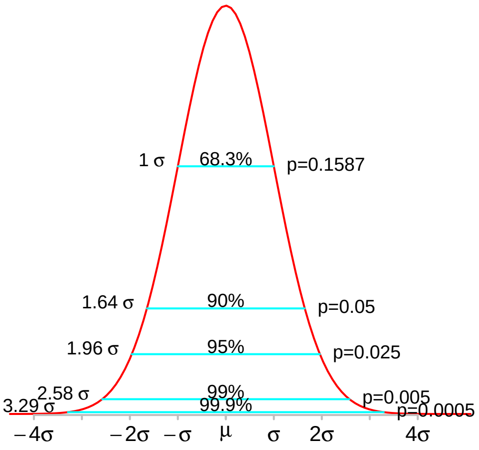


Figure 1.18: Normal distribution with total percentage of values enclosed within a given number of standard deviations. Github–Local

- *Convex/concave* functions: a function $f(x)$ is convex, between a and b, if every chord of the function is above the function. If the chord is always below the function, it is concave; see figure 1.19. The word convex tends to be spoken with a smiley face, while concave induces more of a frown.

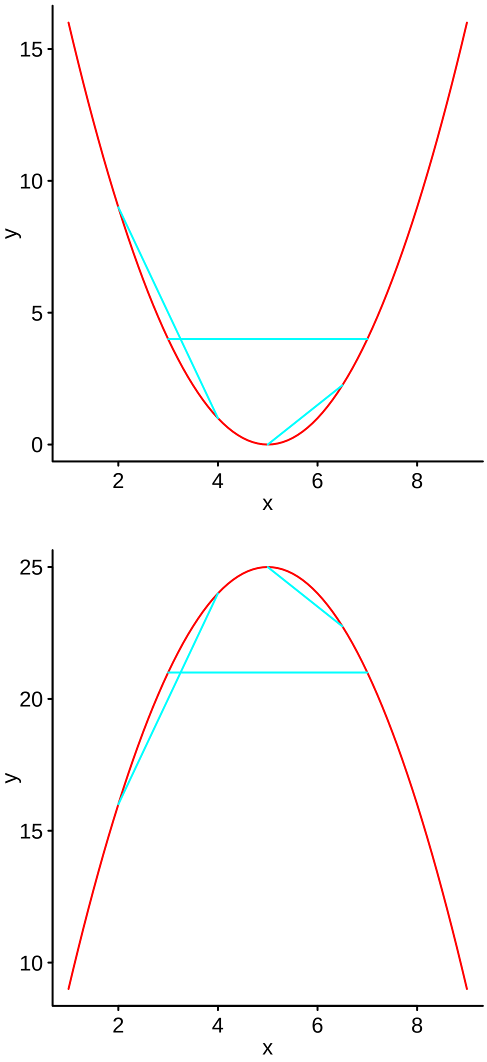


Figure 1.19: Example convex, upper, and concave, lower, functions; lines are three chords of the function. Github–Local

Chapter 2

Human cognition

2.1 Introduction

Software systems are built and maintained by the creative output of human brains, which supply the cognitive effort that directs and performs the activities required. The creation and maintenance of software systems are limited by the cognitive effort available; maximizing the effort delivered requires an understanding of the operating characteristics of the computing platform that produces it.

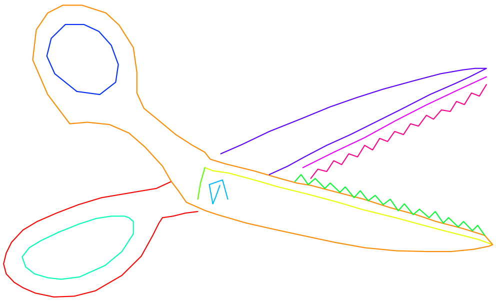


Figure 2.1: Unless cognition and the environment in which it operates closely mesh together, problems may be difficult or impossible to solve; the blades of a pair of scissors need to closely mesh for cutting to occur. Github–Local

Modern humans evolved from earlier humanoids, who in turned evolved from earlier members of the ape family,[806] who in turn evolved from etc., etc. The collection of cognitive characteristics present in the Homo sapien brain is the end-result of a particular sequence of survival pressures that occurred over millions of years of evolutionary history; with the last common ancestor of the great apes, and the line leading to modern humans, living 5 to 7 million years ago,[613] the last few hundred thousand years spent as hunter-gatherers roaming the African savannah, followed by 10,000 years or so having a lifestyle that involved farming crops and raising domesticated animals.

Our skull houses a computing system that evolved to provide responses to problems that occurred in stone-age ecosystems. However, this system is adaptable; neural circuits established for one purpose may be redeployed,[491] during normal development, for different uses, often without losing their original functions, e.g., in many people, learning to read and write involves repurposing the neurons in the ventral visual occipito-temporal cortex (an area of the brain involved in face processing and mirror-invariant visual recognition).[466] Reuse of neural circuitry is a central organizational principle of the brain.[58]

The collection of cognitive characteristics supported by an animal's brain only makes sense in the context of the problems the species had to solve within the environment in which it evolved. Cognition and the environment are like the two blades of a pair of scissors, e.g., figure 2.1, both blades have to mesh together to achieve the desired result.

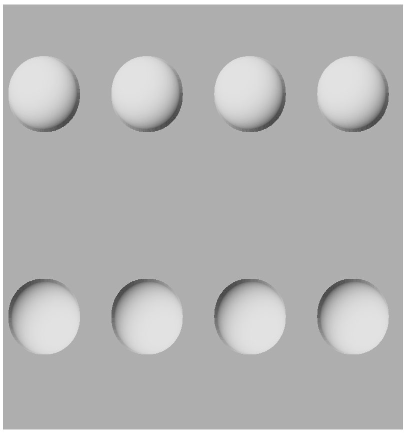


Figure 2.2: The assumption of light shining from above creates the appearance of bumps and pits. Github–Local

- The structure of the natural environment places constraints on optimal performance (an approach to analyzing human behavior known as *rational analysis*),
- Cognitive, perception, and motor operations have their own sets of constraints (an approach known as *bounded cognition*, which targets good-enough performance).

Degraded, or incorrect, performance occurs when cognitive systems have to operate in a situation that violates the design assumptions of the environment they evolved to operate within; the assumptions are beneficial because they simplify the processing of ecologically common inputs. For instance, the human visual system assumes light shines from above, because it has evolved in an environment where this is generally true.[830] A consequence of this assumption of light shining from above is the optical illusion in figure 2.2, i.e., the top row appears as mounds while the lower row appears as depressions.

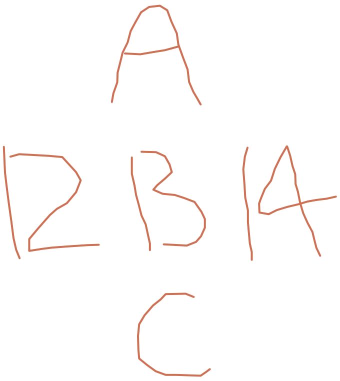


Figure 2.3: Overlearning enables readers to effortlessly switch between interpretations of curved lines. Github–Local

Optical illusions are accepted as curious anomalies of the eye/brain system; there is no rush to conclude that human eyesight is faulty. Failures of the cognitive system to produce answers in agreement with mathematical principles, chosen because they appeal to those making the selection, indicates that the cognitive system has not been tuned by the environment in which it has been successfully to produce answers compatible with the selected mathematical principles.

This book is written from the viewpoint that the techniques used by people to produce software systems should be fitted around the characteristics of the computing platform in our head (the view that developers should aspire to be omnipotent logicians is driven by human self-image, a counter-productive mindset).[i] Builders of bridges do not bemoan the lack of unbreakable materials available to them, they learned how to work within the limitations of the materials available.

Evolutionary psychology[128, 133] is an approach to psychology that uses knowledge and principles from evolutionary biology to help understand the operation of the human mind. Of course physical implementation details, the biology of the brain,[950] also have an impact on psychological performance.

The fact that particular cognitive abilities have benefits in some environments, that outweigh their costs, means they are to be found in a variety of creatures,[158] e.g., numerical competence across the animal kingdom,[1359] and in particular the use of numbers by monkeys[780] and syntax by birds.[1780] A study by Mechner[1237] rewarded rats with food, if they pressed a lever N times (with N taking one of the values 4, 8, 12 or 16), followed by pressing a second lever. Figure 2.4 suggests that rat N1 is making use of an approximate number system. Other examples of non-human cognition are briefly discussed elsewhere, the intent is to show how deep-seated some of our cognitive abilities are, i.e., they may not depend on high-level functionality that is human specific.

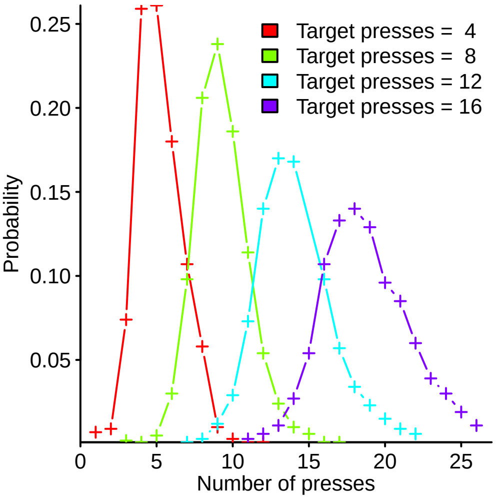


Figure 2.4: Probability that rat N1 will press a lever a given number of times before pressing a second lever to obtain food, when the target count is 4, 8, 12 and 16. Data extracted from Mechner.[1237] Github–Local

Table 2.1 shows a division of human time scales by the kind of action that fits within each interval.

Does the brain contain a collection of modules (each handling particular functionality) or a general purpose processor? This question is the nature vs. nurture debate argument, rephrased using implementation details, i.e., a collection of modules pre-specified by nature or a general purpose processor that is configured by nurture. The term *modularity of mind* refers to a model of the brain[612] containing a general purpose processor attached to special purpose modules that handle perceptual processes (e.g., hearing and sight); the term *massive modularity hypothesis* refers to a model[399] that only contains modules.

Scale (sec)	**Time Units**	**System**	**World (theory)**
10000000	months		
1000000	weeks		Social Band
100000	days		
10000	hours	Task	
1000	10 min	Task	Rational Band
100	minutes	Task	
10	10 sec	Unit task	
1	1 sec	Operations	Cognitive Band
0.1	100 msec	Deliberate act	
0.01	10 msec	Neural circuit	
0.001	1 msec	Neuron	Biological Band
0.0001	100μsec	Organelle	

Table 2.1: Time scales of human action. Based on Newell.[1349]

Consciousness is the tip of the iceberg, most of what goes on in the mind is handled by the unconscious.[331, 424, 1920] Problems that are experienced as easy to solve, may actually involve very complex neural circuitry, and be very difficult to program computers to solve.

The only software engineering activities that could be said to be natural, in that prewired biological structures occur in the brain, involve social activities. The exactitude needed for coding is at odds with the fast and frugal approach of our unconscious mind,[672] whose decisions our conscious mind later does its best to justify.[1920] Reading and writing are not natural in the sense that specialist brain structures have evolved to perform these activities; it is the brain's generic ability to learn that enables this skill to be acquired through many years of deliberate practice.

What are the likely differences in cognitive performance between human males and females? A study by Strand, Deary and Smith[1768] analyzed Cognitive Abilities Test (CAT)

[i] An analysis of the operation of human engineering suggests that attempting to modify our existing cognitive systems is a bad idea,[223] e.g., it is better to rewrite spaghetti code than try to patch it.

scores from over 320,000 school pupils in the UK. Figure 2.5 provides a possible explanation for the prevalence of males at the very competent/incompetent ends of the scale and shows that women outnumber men in the middle competency band (over time differences have remained, but male/female performance ratios have changed[1894]). A model where one sex engages in relatively selective mate choice, produces greater variability in the opposite sex.[820]

A study by Jørgensen and Grimstad[940] asked subjects from three Asian and three east European countries to estimate the number of lines of code they wrote per hour, and the effort needed to implement a specified project (both as part of a project investigating cognitive biases). Both country and gender were significant predictors of the estimates made (see Github–developers/estimation-biases.R).

While much has been written on how society exploits women, relatively little has been written on how society exploits men.[146] There are far fewer women than men directly involved in software engineering.[ii]

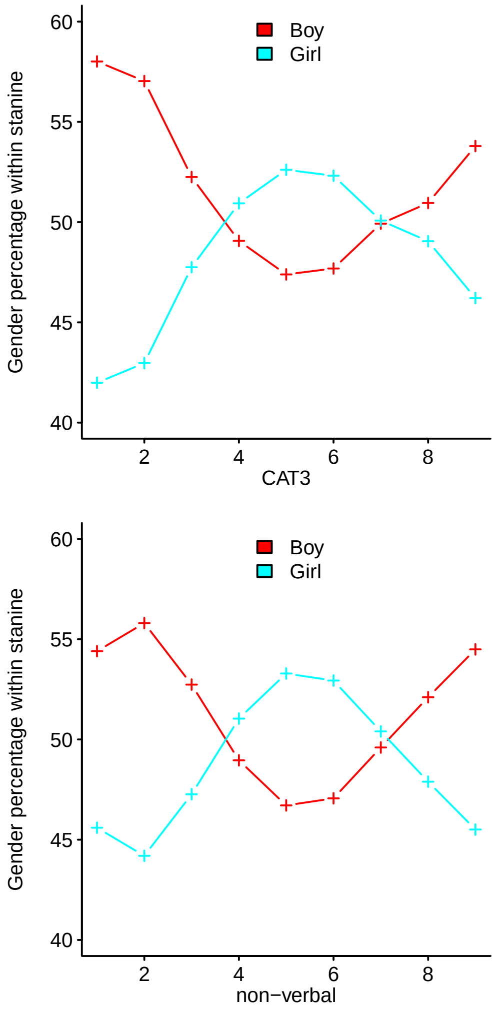


Figure 2.5: Boy/girl (aged 11-12 years) verbal reasoning, quantitative reasoning, non-verbal reasoning and mean CAT score over the three tests; each stanine band is 0.5 standard deviations wide. Data from Strand et al.[1768] Github–Local

2.1.1 Modeling human cognition

Models of human cognitive performance are based on the results of experiments performed subjects drawn almost entirely from Western, Educated, Industrialized, Rich and Democratic (WEIRD) societies.[803, 1634] While this is a problem for those seeking to uncover the cognitive characteristics of humans in general, it is not a problem for software engineering, because those involved have often had a WEIRD society education.

Characteristics of WEIRD people that appear to differ from the general population include:

- WEIRD people are willing to think about, and make inferences about, abstract situations, without the need for having had direct experience; studies[1161] of non-WEIRD people have found they are unwilling to discuss situations where they don't have direct experience,
- when mapping numbers onto space, WEIRD people have been found to use a linear scale for mapping values between one and ten; studies of non-WEIRD people have found they often use a logarithmic scale,[470]
- WEIRD people have been found to have complex, but naive, models of the mechanisms that generate the everyday random events they observe.[1405]

Much of the research carried out in cognitive psychology draws its samples from people between the ages of 18 and 21, studying some kind of psychology degree. There has been discussion[126] on the extent to which these results can be extended to the general populace, however, results obtained by sampling from this subpopulation are likely to be good enough for dealing with software engineering issues.

The reasons why students are not appropriate subjects to use in software engineering experiments, whose results are intended to be applied to professional software developers, are discussed in chapter 13.

Several executable models of the operation of human cognitive processes have been created. The ACT-R model[56] has been applied to a wide range of problems, including learning, the visual interface, perception and action, cognitive arithmetic, and various deduction tasks.

Studies[624] have found a poor correlation between an individual's estimate of their own cognitive ability and measurements of their ability.

Studies of personnel selection[1623] have found that general mental ability is the best predictor of job performance.

Bayesian models of human cognition have been criticised for assuming that performance on a range of tasks is at, or near, optimal;[230] everyday human performance needs to be good enough, and an investment in finding an (near) optimal solution may not be cost effective.

[ii] When your author started working in software development, if there was a woman working on a team, his experience was that she would be at the very competent end of the scale (male/female ratio back then was what, 10/1?). These days, based on my limited experience, women are less likely to be as competent as they once were, but still a lot less likely, than men, to be completely incompetent; is the small number of incompetence women caused by a failure of equal opportunity regulations or because of a consequence of small sample size?

The evidence-accumulation model is used in the cognitive analysis of decision-making. As its name suggests, the model operates by accumulating evidence until the quantity of evidence for one of the options exceeds the threshold needed to trigger its selection. The diffusion model,[1539] and the Linear Ballistic Accumulator (LBA) are currently two leading forms of this model.[514]

Figure 2.6 shows the basic components of evidence-accumulation models; this particular example is of a diffusion model, for a 2-choice decision (i.e., "A" or "B").

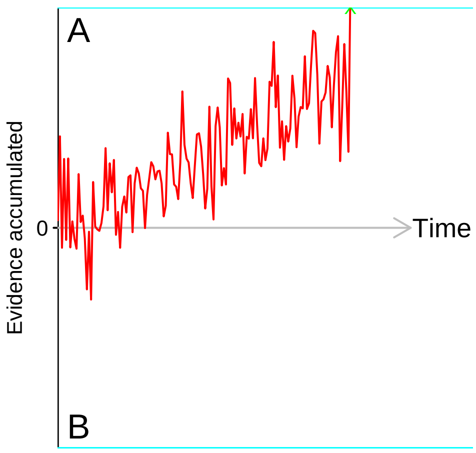


Figure 2.6: Example of the evolution of the accumulation of evidence for option "A", in a diffusion model. Github–Local

The *drift rate* is the average rate at which evidence accumulates in the direction of the correct response; noise in the process creates variability in the response time and the possibility of making an incorrect response. Evidence accumulation may start from a non-zero value.

The distance between the options' decision thresholds, on the evidence axis, impacts response time (increasing distance, increases response time), and error rate (decreasing distance, increases the likelihood that noise can cause sufficient evidence to accumulate to cross the threshold of the incorrect option).

The Linear Ballistic Accumulator model differs from the diffusion model, in that each option has its own register, which holds the evidence accumulated for that option; the first option whose accumulated evidence reaches threshold, is selected (the diffusion model accumulates evidence in a single register, until reaching the threshold of one option).

The measured response time includes what is known as *nondecision time*, e.g., tasks such as information encoding and time taken to execute a response.

Fitting data from experiments[1538] from a lexical decision task,[iii] to a diffusion model, shows that drift rate increases with word frequency, and that some characteristics of non-words (e.g., whether they were similar to words or were random characters) had an impact on the model parameters.

2.1.2 Embodied cognition

Embodied cognition is the theory that many, if not all, aspects of a person's cognitive processing are dependent on, or shaped by, sensory, motor, and emotional processes that are grounded in the features of their physical body.[678]

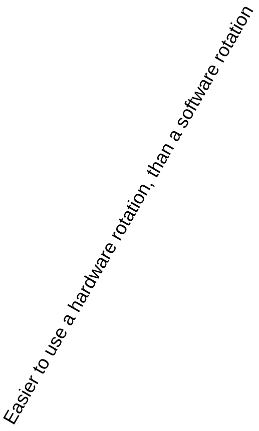


Figure 2.7: Rotating text in the real world; is it most easily read by tilting the head, or rotating the image in the mind? Github–Local

A study by Presson and Montello[1509] asked two groups of subjects to memorize the locations of objects in a room. Both groups were then blindfolded, and then asked to point to various objects; their performance was found to be reasonably fast and accurate. Subjects in one group were then asked to imagine rotating themselves 90°, they were then asked to point to various objects. Their performance was found to be much slower and less accurate. Subjects in the other group were asked to actually rotate 90°; while still blindfolded, they were then asked to point to various objects. The performance of these subjects was found to be as good as before they rotated. These results suggest that mentally keeping track of the locations of objects, a task that might be thought to be cognitive and divorced from the body, is in fact strongly affected by body position.

Tetris players have been found to prefer rotating an item on screen, as it descends, rather than mentally perform the rotation.[998]

A study by Shepard and Metzler[1670] showed subjects pairs of figures and asked if they were the same. Some pairs were different, while others were the same, but had been rotated relative to each other. The results showed a linear relationship between the angle of rotation (needed to verify that two objects were the same), and the time taken to make a matching comparison. Readers might like to try rotating, in their mind, the pair of images in each row of figure 2.8, to find out if they are the same.

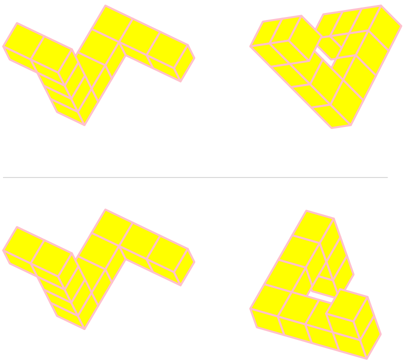


Figure 2.8: Two objects paired with another object that may be a rotated version. Based on Shepard et al.[1670] Github–Local

A related experiment by Kosslyn[1028] showed subjects various pictures and asked questions about them. One picture was of a boat and subjects were asked a question about the front of the boat and then asked a question about the rear of the boat. The response time, when the question shifted from the front to the rear of the boat, was longer than when the question shifted from one about portholes to one about the rear. It was as-if subjects had to scan their image of the boat from one place to another to answer the questions.

Many WEIRD people use a mental left-to-right spatial orientation for the number line. This mental orientation has an embodiment in the *SNARC effect* (spatial numerical association of response codes). Studies[468, 1374] have shown subjects single digit values, and

[iii] Deciding whether a character string is a word.

asked them to make an odd/even decision by pressing the left/right response button with the left-/right-hand; when using the right-hand response time decreases as the value increases (i.e., the value moves from left-to-right along the number line) and when using the left-hand response time decreases as the value decreases. The effect persists when arms are crossed, such that opposite hands are used for button pressing. Figure 2.9 shows the left/right-hand error rate found in a study of the SNARC effect.

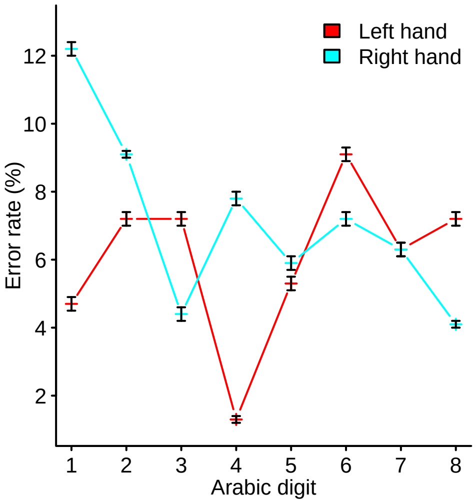


Figure 2.9: Error rate, with standard error, for the left/right-hand from a study of the SNARC effect. Data from Nuerk et al.[1374] Github–Local

2.1.3 Perfection is not cost-effective

Evolution is driven by survival of the fittest, i.e., it is based on relative performance; a consistent flawless performance is not only unnecessary, but a waste of precious resources.

Socially, making mistakes is an accepted fact of life and people are given opportunities to correct mistakes, if that is considered necessary.

For a given task, obtaining information on the kinds of mistakes that are likely to be made (e.g., entering numeric codes on a keyboard[1412]), and modeling the behavior (e.g., subtraction mistakes[1862] made by children learning arithmetic) is very time-consuming, even for simple tasks. Researchers are still working to build good models[1718] for the apparently simple task of text entry.

One technique for increasing the number of errors made by subjects, in an experiment, is to introduce factors that will increase the likelihood of mistakes being made. For instance, under normal circumstances the letters/digits viewed by developers are clearly visible, and the viewing time is not constrained; in experiments run under these conditions subjects make very few errors. To obtain enough data to calculate letter similarity/confusability, studies[1304] have to show subjects images of single letters/digits that have been visually degraded, or to limit the amount of time available to make a decision, or both, until a specified error rate is achieved.[1819] While such experiments may provide the only available information, on the topic of interest, their ecological validity has to be addressed (compared to say asking subjects to rate pairs of letters for similarity[1692]).

How often do people make mistakes?

A lower bound on human error rate, when performing over an extended period, is probably that of astronauts in space; making an error during a space mission can have very serious consequences, and there is a huge investment in astronaut training. NASA maintains several databases of errors made by operators during simulation training and actual missions; human error rates, for different missions, of between $1.9 \cdot 10^{-3}$ and $1.05 \cdot 10^{-4}$ have been recorded.[310]

Touch typists, who are performing purely data entry:[1207] with no error correction 4% (per keystroke), typing nonsense words (per word) 7.5%.

A number of human reliability analysis methods[311] for tasks in safety critical environments are available. The Cognitive Reliability Error Analysis Model (CREAM) is widely used; Calhoun et al[284] work through a calculation of the probability of an error during the International Space Station ingress procedure, using CREAM. The HEART method[1944] is another.

How do people respond when their mistakes are discovered?

A study by Jørgensen and Moløkken[943] interviewed employees, from one company, with estimation responsibilities. Analysis of the answers showed a strong tendency for people to perceive factors outside their control as important contributing factors for inaccurate estimates, while factors within their control were typically cited as reasons for accurate estimates.

2.2 Motivation

Motivation is a powerful influence on an individuals' priorities. The desire to complete an immediate task can cause a person to alter their priorities in a way they might not choose in more relaxed circumstances. A study by Darley and Batson[430] asked subjects (theological seminary students) to walk across campus to deliver a sermon. Some subjects were told that they were late, and the audience was waiting, the remainder were not told this. Their journey took them past a victim moaning for help in a doorway. Only 10% of subjects who thought they were late stopped to help the victim, while 63% of the other subjects stopped to help. These results do not match the generally perceived behavior pattern of theological seminary students.

What motivations do developers have while working on software systems? Possible motivations include those that generally apply to people (e.g., minimizing cognitive effort, see section 2.2.2), apply to company employees (e.g., seeking promotion, or doing a job because it pays for personal interests), apply to those involved in creative activities (e.g., organizing activities to maximize the pleasure obtained from work).

Hedonism is an approach to life that aims to maximise personal pleasure and happiness (see section 11.3.5, for an analysis of a study investigating the importance of fun as a motivation for software development). Many theories of motivation take as their basis that people intrinsically seek pleasure and avoid pain, i.e., they are driven by *hedonic motivation*.

People involved in work that requires creativity can choose to include personal preferences and desires in their decision-making process, i.e., they are subject to hedonic motivation. Todate, most research on hedonic motivation research has involved studies of consumer behavior.

The theory of *regulatory focus theory* is based around the idea that people's different approaches to pleasure and pain influences the approach they take towards achieving an end-state (or, end-goal). The theory contains two end states, one concerned with aspirations and accomplishments (a *promotion focus*), and the other concerned with attainment of responsibilities and safety (a *prevention focus*).

A promotion focus is sensitive to presence and positive outcomes, seeks to insure hits and insure against errors of omission. A prevention focus is sensitive to absence and negative outcomes, seeks to insure against correct rejections and insure against errors of commission.

People are able to exercise some degree of executive control over the priorities given to cognitive processes, e.g., deciding on speed/accuracy trade-offs. Studies[618] have found that subjects with a promotion focus will prefer to trade-off accuracy for speed of performance, and those with a prevention focus will trade-off speed for improved accuracy.

The concept of *Regulatory fit*[816] has been used to explain why people engage more strongly with some activities and "feel right" about it (because the activity sustains, rather than disrupts, their current motivational orientation or interests).

2.2.1 Built-in behaviors

Early researchers, investigating human behavior, found that people do not always respond in ways that are consistent with the mathematically optimal models of behavior that had been created. The term *cognitive bias* has become the umbrella term used to describe such behavior.

Evolution selects for traits that provide an organism with advantages within its ecosystem; the failure of mathematical models, created by researchers, to predict human responses, is a failure of researchers to understand human behavior, not a failure of people to behave according to these models. Researchers are starting to create models[1128] where, what were once though to be cognitive biases, are actually ecologically rational uses of cognitive resources.

Evidence-based software engineering has to take into account whatever predispositions come included, as standard, with every human brain. The fact that many existing models of human behavior poorly describe real-life behaviors means that extreme caution is needed when attempting to model developer behaviors.

Anchoring bias and *confirmation bias* are two commonly discussed cognitive biases.

Anchoring: behavior where people assign too much weight to the first piece of information they obtain, relating to the topic at hand.

A study by Jørgensen and Sjøberg[945] asked professionals and students to estimate the development effort for a project, with one group of subjects being given a low estimate from the *customer*, a second group a low estimate (and no rationale), and the third group no *customer* estimate. The results (see Github–developer/anchor-estimate.R) showed that estimates from subjects given a high/low customer estimate were much higher/lower than subjects who did not receive any customer estimate.

A study by Jørgensen and Grimstad[940] asked subjects to estimate the number of lines of code they wrote per hour, with subjects randomly split into two groups; one anchored with the question: “Did you on average write more or less than 1 Line of Code per work-hours in your last project?”, and the other with: “Did you on average write more or less than 200 Lines of Code per work-hours in your last project?” Fitting a regression model to the results showed that the form of the question changed the value estimated by around 72 lines (sd 10). Github–developers/estimation-biases.R

Confirmation bias: occurs when ambiguous evidence is interpreted as (incorrectly) confirming a person’s current beliefs about the world. For instance, developers interpreting program behavior as supporting their theory of how it operates, or using the faults exhibited by a program to conform their view that it was poorly written.

When shown data from a set of observations, a person might propose a set of rules that the processes generating the data adhere to. Given the opportunity to test proposed rules, what strategy are people likely to use?

A study by Wason,[1910] which became known as the *2–4–6 Task*, asked subjects to discover a rule known to the experimenter; subjects’ guessed a rule, told it to the experimenter, who told them whether the answer was correct. For instance, on being informed that the sequence 2–4–6 was an instance of the experimenter’s rule, possible subject rules might be “two added each time” or “even numbers in order of magnitude”, when perhaps the actual rule was “three numbers in increasing order of magnitude”.

An analysis of the rules created by subjects found that most were test cases designed to confirm a hypothesis about the rule (known as a *positive test strategy*), with few test cases attempting to disconfirm a hypothesis. Some subjects declared rules that were mathematically equivalent variations of rules they had already declared.

The use of a positive test strategy has been claimed to be a deficiency, because of the work of Popper,[1486] who proposed that scientists should perform experiments designed to disprove their hypothesis. Studies[1612] of the hypothesis testing strategies used by scientists found that positive testing is the dominant approach.

An analysis by Klayman and Ha[1007] investigated the structure of the problem subjects were asked to solve. The problem is a search problem, find the experiment’s rule, and in some environments a positive test strategy is more effective for solving search problems, compared to a negative test strategy. Figure 2.10 shows the five possible ways in which the experimenter’s rule and subject’s hypothesis can ovcerlap.

A positive test strategy is more effective when the sought after rule describes a minority case, i.e., there are more cases not covered by the rule, or when the hypothesized rule includes roughly as many cases as the actual rule, that is, the hypothesized rule is about the right size. Klayman and Ha claimed these conditions hold for many real-world rule search problems, and a positive test strategy is therefore an adaptive strategy; the real-worldness claim continues to be debated.[1339]

The implementation of positive and negative test cases is discussed in section 6.6.2.3.

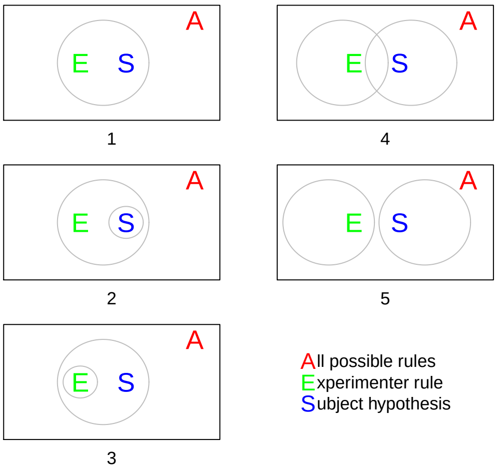


Figure 2.10: The five possible ways in which experimenter’s rule and subject’s rule hypothesis can overlap, in the space of all possible rules; based on Klayman et al.[1007] Github–Local

2.2.2 Cognitive effort

When attempting to solve a problem, a person’s cognitive system (consciously and unconsciously) makes cost/accuracy trade-offs. The details of how it forms an estimate of the value, cost and risk associated with an action, and carries out the trade-off analysis is not known (various models have been proposed[725]). An example of the effects of these trade-offs is provided by a study by Fu and Gray,[626] where subjects had to copy a pattern of colored blocks (on a computer-generated display). Remembering the color of the block to be copied, and its position in the target pattern, created a memory effort; a perceptual-motor effort was introduced by graying out the various areas of the display, where the colored blocks were visible; these grayed out areas could be made temporarily visible using various combinations of keystrokes and mouse movements. Subjects had the choice of expending memory effort (learning the locations of different colored blocks) or perceptual-motor effort (using keystrokes and mouse movements to uncover different areas of the display). A subject’s total effort is the sum of perceptual motor effort and memory storage and recall effort.

The subjects were split into three groups; one group had to expend a low effort to uncover the grayed out areas, the second acted as a control, and the third had to expend a high effort to uncover the grayed out areas. The results showed that the subjects who had to expend a high perceptual-motor effort, uncovered grayed out area fewer times than the other two groups. These subjects also spent longer looking at the areas uncovered, and moved more colored blocks between uncoverings. The subjects faced with a high perceptual-motor effort reduced their total effort by investing in memory effort. Another consequence of this switch of effort investment, to use of memory, was an increase in errors made.

What are the physical processes that generate the feeling of mental effort? The processes involved remain poorly understood;[1668] proposals include: metabolic constraints (the brain accounts for around 20% of heart output, and between 20% to 25% of oxygen and glucose requirements), but the energy consumption of the visual areas of the brain while watching television are higher than the consumption levels of those parts of the brain associated with difficult thinking, and that feeling of mental effort is the body's reaction to concentrated thinking is to try to conserve energy, for use in other opportunities that could arise in the immediate future, by creating a sense of effort.[1047]

2.2.3 Attention

Most sensory information received by the brain does not require conscious attention, and can be handled by the unconscious. Conscious attention is like a spotlight shining cognitive resources on a chosen area. In today's world, there is often significantly more information available to a person than they have available attention resources, WEIRD people live in an attention economy.

People can direct attention to their internal thought processes and memories. For instance, read the bold text in the following paragraph:

Somewhere **Among** hidden **the** in **most** the **spectacular** Rocky Mountains **cognitive** near **abilities** Central City **is** Colorado **the** an **ability** old **to** miner **select** hid **one** a **message** box **from** of **another.** gold. **We** Although **do** several **this** hundred **by** people **focusing** have **our** looked **attention** for **on** it, **certain** they **cues** have **such** not **as** found **type** it **style**.

What do you remember from the non-bold text? Being able to make a decision to direct conscious attention to inputs matching a given pattern is a technique for making efficient use of limited cognitive resources.

Much of the psychology research on attention has investigated how inputs from our various senses are handled. It is known that they operate in parallel, and at some point there is a serial bottleneck, beyond which point it is not possible to continue processing input stimuli in parallel. The point at which this bottleneck occurs is a continuing subject of debate; there are early selection theories, late selection theories, and theories that combine the two.[1429]

A study by Rogers and Monsell[1577] investigated the impact of task switching on subject performance. Subjects were split into three groups; one group was given a letter classification task (is a letter a consonant or vowel), the second group a digit classification task (is the digit odd or even), and the third group had to alternate tasks (various combinations were used) between letter, and digit classification. The results found that having to alternate tasks slowed response times by 200 to 250 ms and the error rates went up from between 2-3%, to between 6.0-7.6%. A study by Altmann[45] found that when the new task had many features in common with the previous task (e.g., switching from classifying numbers as odd or even, to classifying them as less than or greater than five) the memories for the related tasks interfered, causing a reduction in subject reaction time, and an increase in error rate.

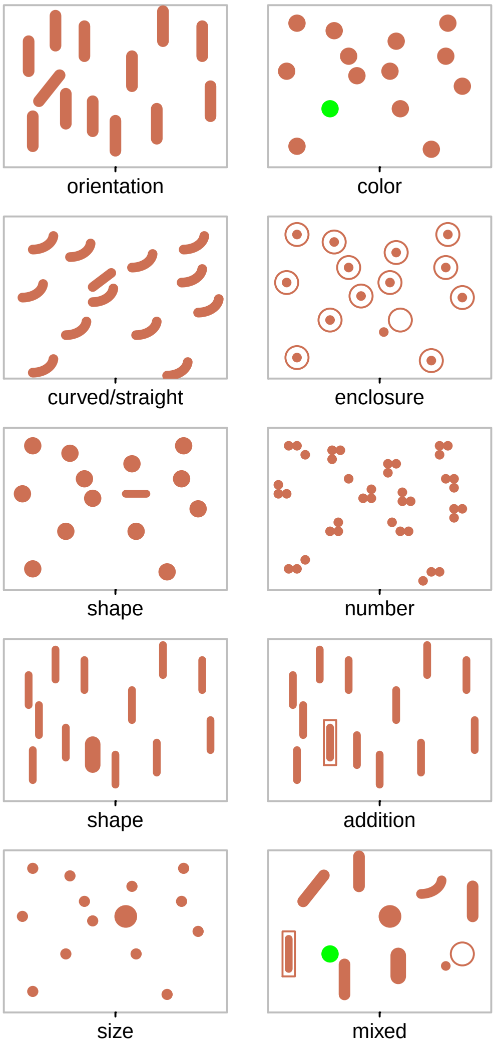


Figure 2.11: Examples of features that may be preattentively processed; when items having distinct features are mixed together, individual items no longer jump out. Based on example in Ware.[1908] Github–Local

2.3 Visual processing

Visual processing is of interest to software development because it consumes cognitive resources, and is a source of human error. The 2-D image falling on the retina does not contain enough information to build the 3-D model we *see*; the mind creates this model by making assumptions about how objects in our environment move and interact.[830]

The perceptual systems of organisms have evolved to detect information in the environment that is relevant to survival, and ignore the rest. The relevant information is about opportunities *afforded* by the world.

The human visual system contains various hardware systems dedicated to processing visual information; low level details are preprocessed to build structures having a higher level of abstraction, e.g., lines. Preattentive processing, so-called because it occurs before conscious attention,[1522] is automatic and apparently effortless. This preprocessing causes items to appear to *pop-out* from their surroundings. Figure 2.11 shows examples of items that pop-out at the reader.

Preattentive processing is independent of the number of distractors; a search for the feature takes the same amount of time whether it occurs with one, five, ten, or more other distractors. However, it is only effective when the features being searched for are relatively rare. When a display contains many, distinct features (the mixed category in figure 2.11), the *pop-out* effect does not occur.

The *Gestalt laws of perception* (“gestalt” means “pattern” in German, also known as the *laws of perceptual organization*)[1893] are based on the underlying idea that the whole is different from the sum of its parts. These so-called *laws* do not have the rigour expected of a scientific law, and some other term ought to be used, e.g., principle. The Gestalt principles are preprogrammed (i.e., there is no conscious cognitive cost). The following are some commonly occurring principles:

- continuity, also known as *good continuation*: Lines and edges that can be seen as smooth and continuous are perceptually grouped together; figure 2.12 upper left,
- closure: elements that form a closed figure are perceptually grouped together; figure 2.12 upper right,
- symmetry: treating two, mirror image lines as though they form the outline of an object; figure 2.12 second row down. This effect can also occur for parallel lines,
- proximity: elements that are close together are perceptually grouped together; figure 2.12 second row up,
- similarity: elements that share a common attribute can be perceptually grouped together; figure 2.12 lower row,
- other: principles include grouping by connectedness, grouping by common region, and synchrony.[1421]

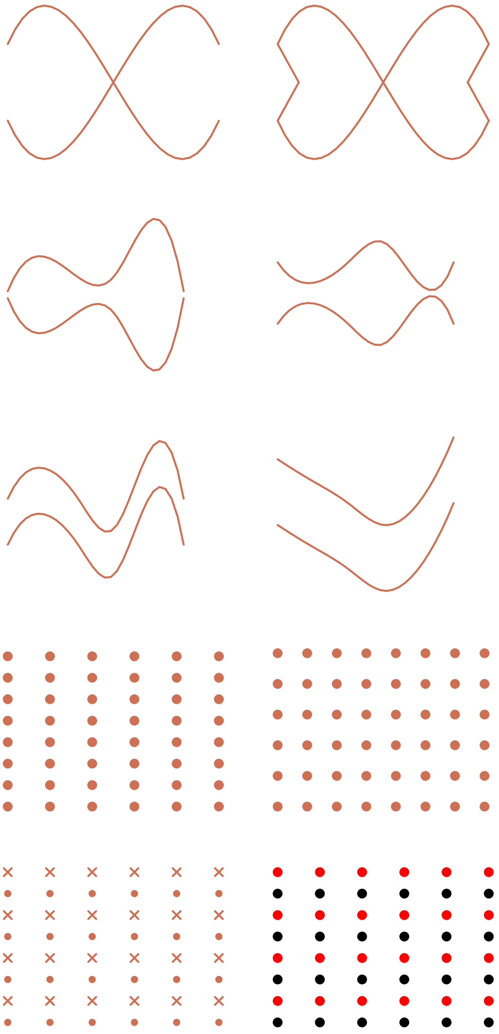


Figure 2.12: Continuity—upper left plot is perceived as two curved lines; Closure—when the two perceived lines are joined at their end (upper right), the perception changes to one of two cone-shaped objects; Symmetry and parallelism—where the direction taken by one line follows the same pattern of behavior as another line; Proximity—the horizontal distance between the dots in the lower left plot is less than the vertical distance, causing them to be perceptually grouped into lines (the relative distances are reversed in the right plot); Similarity—a variety of dimensions along which visual items can differ sufficiently to cause them to be perceived as being distinct; rotating two line segments by 180°does not create as big a perceived difference as rotating them by 45°. Github–Local

Visually grouping the elements in a display, using these principles, is a common human trait. However, different people can make different choices, when perceptually grouping of the same collection of items. Figure 2.13 shows items on a horizontal line, which readers may group by shape, color, or relative proximity. A study by Kubovy and van den Berg[1037] created a model that calculated the probability of a particular perceptual grouping (i.e., shape, color, or proximity in two dimensions) being selected for a given set of items.

Studies[1066] have found that when mapping the prose specification of a mathematical relationship to a formula, the visual proximity of applicable names has an impact on the error rate.

A study by Palmer, Horowitz, Torralba and Wolfe[1420] investigated the distribution of subject response times when searching for targets having varying degrees of distinctiveness from the surrounding items, and with varying numbers of surrounding items. Subjects each completed 400 trials locating a target among items sharing some visual characteristics with the target; in 50% of trials the target was not present. Figure 2.14 shows examples of the three tasks used, each containing a target that had various levels of distinctiveness from the other surrounding items.

Figure 2.15 shows the mean response time for each subject, when target was present (+ character) or absent (o character) from the display, along with fitted regression lines (solid when target present, dashed when absent).

Depending on the visual characteristic, the search for an item may be sequential (e.g., shape, when many items share the same shapes), or have some degree of parallelism (e.g., color, when items have one of a few distinct colors); for an example, see figure 8.35.

2.3.1 Reading

Building software systems involves a significant amount of reading. Research on the cognitive processes involved in reading prose, written in human languages, has uncovered the processes involved and various models have been built.[1543]

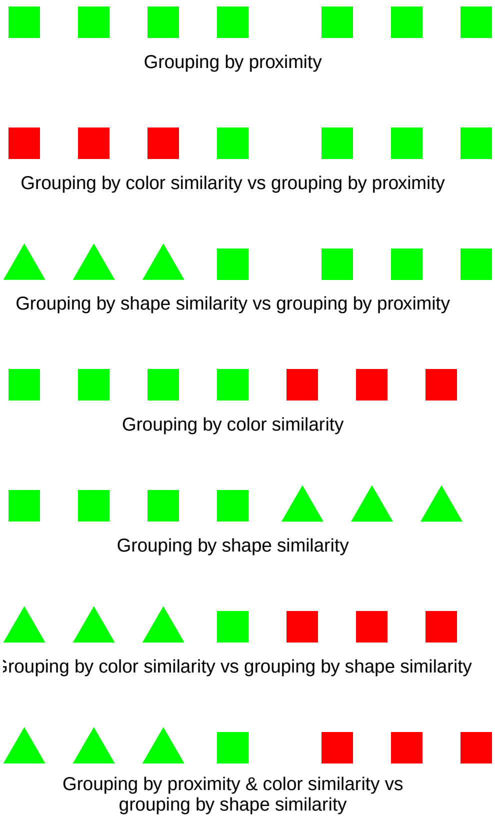

Grouping by proximity
Grouping by color similarity vs grouping by proximity
Grouping by shape similarity vs grouping by proximity
Grouping by color similarity
Grouping by shape similarity
rouping by color similarity vs grouping by shape similarity
Grouping by proximity & color similarity vs grouping by shape similarity

Figure 2.13: Perceived grouping of items on a line may be by shape, color or proximity. Based on Kubovy et al.[1037] Github–Local

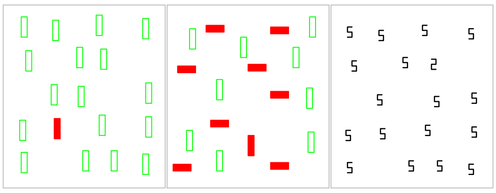


Figure 2.14: Examples of the three tasks subjects were asked to solve. Left (RV GV): solid red rectangle having same alignment with outline green rectangle, middle (RV RHGV): solid vertical rectangle among solid horizontal rectangles and outlined vertical green rectangles, and right (2 5): digital 2 among digital 5s. Adapted from Palmer et al.[1420] Github–Local

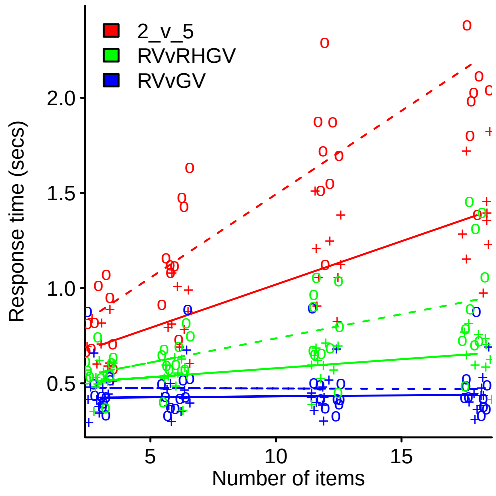


Figure 2.15: Average subject response time to find a target in an image containing a given number of items (x-axis), when target present (+ and solid line) and absent (o and dashed line); lines are fitted regression models. Data from Palmer et al.[1420] Github–Local

When reading prose, a person's eyes make short rapid movements, known as *saccades*, taking 20 ms to 50 ms to complete; a saccade typically moves the eyes forward 6 to 9 characters. No visual information is extracted during a saccade, and readers are not consciously aware of them. Between saccades the eyes are stationary, typically for 200 ms to 250 ms, these stationary periods are known as *fixations*. A study[1466] of consumer eye movements, while comparing multiple brands, found a fixation duration of 354 ms when subjects were under high time pressure and 431 ms when under low time pressure.

Individual readers can exhibit considerable variations in performance, a saccade might move the eyes by one character, or 15 to 20 characters; fixations can be shorter than 100 ms or longer than 400 ms (there is also variation between languages[1404]). The content of fixated text has a strong effect on performance.

The eyes do not always move forward during reading—10% to 15% of saccades move the eyes back to previous parts of the text. These backward movements, called *regressions*, are caused by problems with linguistic processing (e.g., incorrect syntactic analysis of a sentence), and oculomotor error (e.g., the eyes overshooting their intended target).

Saccades are necessary because the eyes' field of view is limited. Light entering an eye hits light-sensitive cells in the retina, where cells are not uniformly distributed. The visual field (on the retina) can be divided into three regions: foveal (the central 2°, measured from the front of the eye looking toward the retina), parafoveal (extending out to 5°), and peripheral (everything else); see figure 2.16. Letters become increasingly difficult to identify as their angular distance from the center of the fovea increases.

During the fixation period, two processes are active: identifying the word (or sequence of letters forming a partial word), and planning the next saccade (when to make it and where to move the eyes). Reading performance is speed limited by the need to plan and perform saccades (removing the need to saccade, by presenting words at the same place on a display, results in a threefold speed increase in reading aloud, and a two-fold speed increase in silent reading). The time needed to plan and perform a saccade is approximately 180 ms —200 ms (known as the *saccade latency*), which means the decision to make a saccade occurs within the first 100 ms of a fixation.

The contents of the parafoveal region are partially processed during reading, and this increases a reader's perceptual span. When reading words written using alphabetic characters, the perceptual span extends from 3 to 4 characters on the left of fixation, to 14 to 15 letters to the right of fixation. This asymmetry in the perceptual span is a result of the direction of reading, attending to letters likely to occur next is a cost effective use of resources. Readers of Hebrew (which is read right-to-left) have a perceptual span that has opposite asymmetry (in bilingual Hebrew/English readers the direction of the asymmetry depends on the language being read, showing the importance of attention during reading).[1542]

Characteristics used by the writing system affect the asymmetry of the perceptual span and its width, e.g., the span can be smaller for Hebrew than English (Hebrew words can be written without the vowels, requiring greater effort to decode and plan the next saccade). It is also much smaller for writing systems that use ideographs, such as Japanese (approximately 6 characters to the right) and Chinese.

The perceptual span is not hardwired, but is attention-based. The span can become smaller when the fixated words become difficult to process. Also, readers extract more information in the direction of reading when the upcoming word is highly predictable (based on the preceding text).

Models of reading that have achieved some level of success include: Mr. Chips[1097] is an ideal-observer model of reading (it is not intended to model how humans read, but to establish the pattern of performance, when optimal use is made of available information), which attempts to calculate the distance, in characters, of the next saccade. It combines information from visual data obtained by sampling the text through a retina, lexical knowledge of words and their relative frequencies, and motor knowledge of the statistical accuracy of saccades, and uses the optimization principle of entropy minimization. The SWIFT model[536] attempts to provide a realistic model of saccade generation, while E-Z Reader[1483] attempts to account for how cognitive and lexical processes influence the eye movements of skilled readers (it can handle the complexities of *garden path* sentences[1549][iv]).

[iv] In "Since Jay always jogs a mile seems like a short distance." readers experience a disruption that is unrelated to the form or meaning of the individual words; the reader has been led down the syntactic garden path of initially parsing the sentence such that a `mile` is the object of `jogs`, before realizing that a `mile` is the subject of `seems`.

English text is read left to right, on lines that go down the page. The order in which the components of a formula are read depends on its contents, with the visual processing of subexpressions by experienced users driven by the mathematical syntax,[901, 902] with the extraction of syntax happening in parallel.[1624]

A study by Pelli, Burns, Farell, and Moore[1443] found that 2,000 to 4,000 trials were all that was needed for novice readers to reach the same level of efficiency as fluent readers in the letter-detection task (efficiency was measured by comparing human performance compared to an ideal observer). They tested subjects aged 3 to 68 with a range of different (and invented) alphabets (including Hebrew, Devanagari, Arabic and English). Even fifty years of reading experience, over a billion letters, did not improve the efficiency of letter detection. They also found this measure of efficiency was inversely proportional to letter perimetric complexity (defined as: inside and outside perimeter squared, divided by *ink* area).

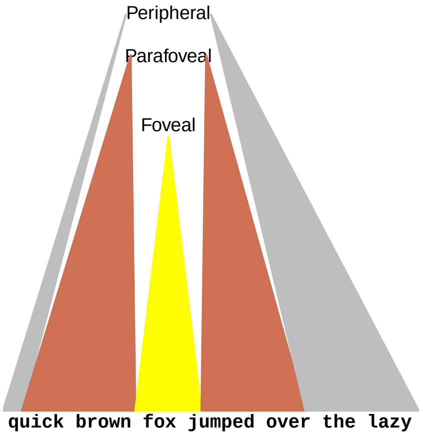


Figure 2.16: The foveal, parafoveal and peripheral vision regions when three characters visually subtend 3°. Based on Schotter et al.[1631] Github–Local

The choice of display font is a complex issue. The use of Roman, rather than Helvetica (or serif vs. sans serif), is often claimed to increase reading speed and comprehension. The issues involved in selecting fonts are covered in a report detailing “Font Requirements for Next Generation Air Traffic Management Systems”.[252]

Vision provides information about what people are thinking about; gaze follows shifts of visual attention. Tracking a subject’s eye movements and fixations, when viewing images or real-life scenes, is an established technique in fields such as marketing and reading research. This technique is now starting to be used in research of developer code reading; see figure 2.17.

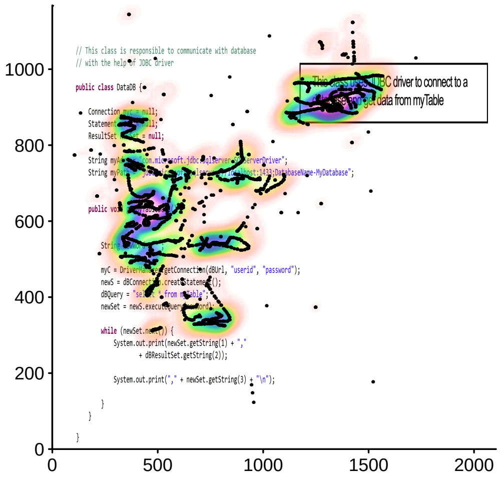


Figure 2.17: Heat map of one subject’s cumulative fixations (black dots) on a screen image. Data kindly provided by Ali.[30] Github–Local

2.4 Memory systems

Memory evolved to supply useful, timely information to an organism’s decision-making systems,[1009] subject to evolutionary constraints.[1332] Memory subsystems are scattered about the brain, with each subsystem/location believed to process and store distinct kinds of information. Figure 2.18 hows a current model of known long-term memory subsystems, along with the region of the brain where they are believed to operate.

Declarative memory has two components, each processing specific instances of information, one handles facts about the world, while the other, episodic memory, deals with events (i.e., the capacity to experience an event in the context in which it occurred; it is not known if non-human brains support episodic memory). We are consciously aware of declarative memory, facts and events can be consciously recalled; it is the kind of memory that is referred to in everyday usage as *memory*. Declarative memory is representational, it contains information that can be true or false.

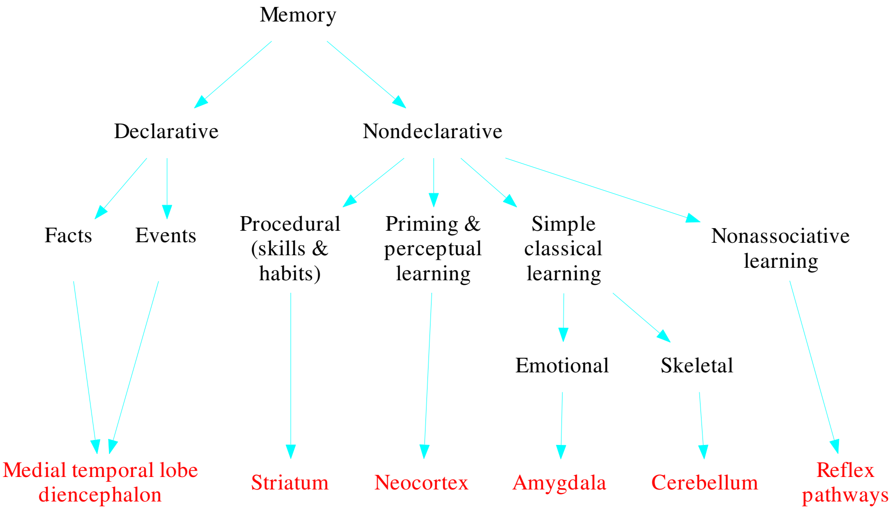


Figure 2.18: Structure of mammalian long-term memory subsystems; brain areas in red. Based on Squire et al.[1732]

Nondeclarative memory (also known as *implicit memory*) extracts information from recurring instances of an experience, to form skills (e.g., speaking a language) and habits, simple forms of conditioning, priming (i.e., response to a stimulus is modified by a preceding stimulus;[1553] an advantage in a slowly changing environment, where similar events

3 3 3 3
a a a a a a
8 8 8
z z
1 1
t t t t
6 6 6 6 6

are likely to occur on a regular basis), and perceptual learning (i.e., gradual improvement in the detection or discrimination of visual stimuli with practice).

Information in nondeclarative memory is extracted through unconscious performance, e.g., riding a bike. It is an unconscious memory and is not available for conscious recall; information use requires reactivation of the subsystem where the learning originally occurred.

These subsystems operate independently and in parallel, the parallelism creates the possibility of conflicting signals being fed as inputs to higher level systems. For instance, a sentence containing the word blue may be misinterpreted because information about the word, and the color in which it appears, green, is returned by different memory subsystems (known as the *Stroop effect*).

A Stroop-like effect has also been found to occur with lists of numbers. Readers might like to try counting the number of characters occurring in each row in the outside margin. The effort of counting the digit sequences is likely to be greater, and more error prone, than for the letter sequences.

Studies[1434] have found that when subjects are asked to enumerate visually presented digits, the amount of Stroop-like interference depends on the arithmetic difference between the magnitude of the digits used, and the quantity of those digits displayed. Thus, a short, for instance, list of large numbers is read more quickly, and with fewer errors, than a short list of small numbers. Alternatively a long list of small numbers (much smaller than the list length) is read more quickly, and with fewer errors, than a long list of numbers where the number has a similar magnitude than the length of the list.

Making use of patterns that can be seen in the environment is one technique for reducing memory load.

Studies have found that people organize their memory for objects within their visual field according to the relative positions of the objects. A study by McNamara, Hardy, and Hirtle[1231] gave subjects two minutes to memorize the location of objects on the floor of a room; see figure 2.19, upper plot. The objects were then placed in a box, and subjects were asked to replace the objects in their original position; the memorize/recall cycle was repeated, using the same layout, until the subject could place all objects in their correct position.

The order in which each subject recalled the location of objects was mapped to a hierarchical tree (one for each subject). The resulting trees (see figure 2.19, lower plot) showed how subjects' spatial memory of the objects had an organization based on spatial distance between items.

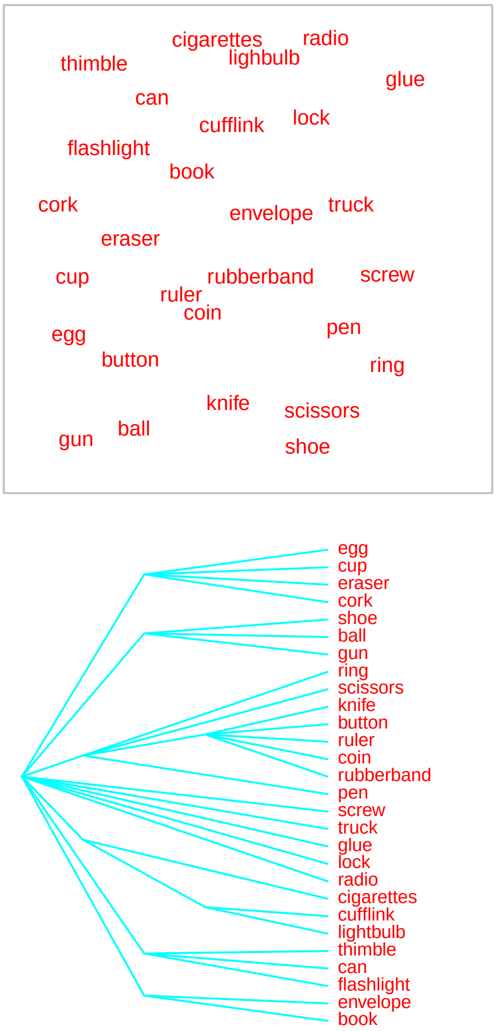


Figure 2.19: Example object layout, and the corresponding ordered tree produced from the answers given by one subject. Data extracted from McNamara et al.[1231] Github–Local

Other research on the interaction between human memory and software development includes: a study[44] which built a computational process model, based on SOAR, and fitted it to 10.5 minutes of programmer activity (debugging within an `emacs` window); the simulation was used to study the memories built up while trying to solve a problem.

8704
2193
3172
57301
02943
73619
659420
402586
542173
6849173
7931684
3617458
27631508
81042963
07239861
578149306
293486701
721540683
5762083941
4093067215
9261835740

2.4.1 Short term memory

As its name implies, *short term memory* (STM) is a memory system that can hold information for short periods of time. Short term memory is the popular term for what cognitive psychologists call *working memory*, named after its function, rather than the relative duration holds information. Early researchers explored its capacity, and a paper by Miller[1267] became the citation source for the now-famous 7±2 rule (Miller did not propose 7±2 as the capacity of STM, but simply drew attention to the fact that this range of values fitted the results of several experiments). Things have moved on in the 65+ years since the publication of his paper,[917] with benchmarks now available for evaluating models of STM.[1380]

Readers might like to try measuring their STM capacity, using the list of numbers in the outside margin. Any Chinese-speaking readers can try this exercise twice, using the English and Chinese words for the digits (use of Chinese should enable readers to apparently increase the capacity of their STM). Slowly and steadily read the digits in a row, out loud. At the end of each row, close your eyes and try to repeat the sequence of digits in the same order. If you make a mistake, go on to the next row. The point at which you cannot correctly remember the digits in any two rows, of a given length, indicates your capacity limit, i.e., the number of digits in the previous two rows.

The performance impact of reduced working memory capacity can be shown by having people perform two tasks simultaneously. A study by Baddeley[107] measured the time taken to solve a simple reasoning task (e.g., $B \rightarrow A$, question: "A follows B" True or False?), while remembering a sequence of digits (the number of digits is known as the *digit load*). Figure 2.20 shows response time (left axis) and percentage of incorrect answers (right axis) for various digit loads.

26
12

Measuring memory capacity using lists of digits relies on a variety of assumptions, such as assuming all items consume the same amount of memory resources (e.g., digits and letters are interchangeable), that relative item ordering is implicitly included in the measurement and that individual concepts are the unit of storage. Subsequent studies have completely reworked models of STM. What the preceding exercise measured was the amount of *sound* that could be held in STM. The spoken sound used to represent digits in Chinese is shorter than in English, and using Chinese should enable readers to maintain information on more digits (average 9.9[842]) using the same amount of sound storage. A reader using a language in which the spoken sound of digits is longer, can maintain information on fewer digits, e.g., average 5.8 in Welsh,[534] and the average for English is 6.6.

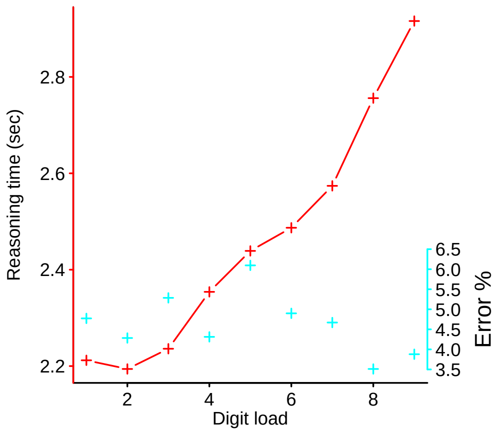


Figure 2.20: Response time (left axis) and error percentage (right axis) on reasoning task with a given number of digits held in memory. Data extracted from Baddeley.[107] Github–Local

Observations of a 7±2 capacity limit derive from the number of English digits spoken in two seconds of sound[109] (people speak at different speeds, which is one source of variation included in the ±2; an advantage for fast talkers). The two seconds estimate is based on the requirement to remember items, and their relative order; the contents of STM do not get erased after two seconds, this limit is the point at which degradation of its contents start to become noticeable.[1303] If recall of item order is not relevant, then the limit increases because loss of this information is not relevant.

Studies[1381] involving multiple tasks have been used to distinguish the roles played by various components of working memory (e.g., storage, processing, supervision and coordination). Figure 2.21 shows the components believed to make up working memory, each with its own independent temporary storage areas, each holding and using information in different ways.

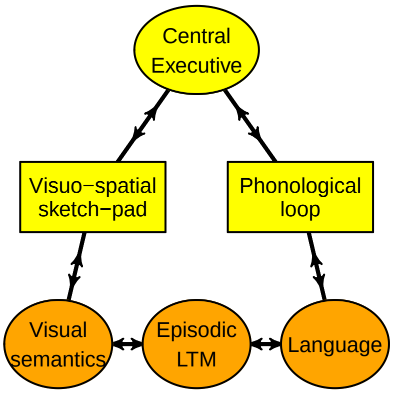


Figure 2.21: Major components of working memory: working memory in yellow, long-term memory in orange. Based on Baddeley.[108] Github–Local

The central executive is assumed to be the system that handles attention, controlling the phonological loop, the visuo-spatial sketch pad, and the interface to long-term memory. The central executive needs to remember information while performing tasks, such as text comprehension and problem-solving. It has been suggested that the focus of attention is capacity-limited, but that the other temporary storage areas are time-limited (without attention to rehearse them, they fade away).[406]

Visual information held in the visuo-spatial sketch pad decays very rapidly. Experiments have shown that people can recall four or five items immediately after they are presented with visual information, but that this recall rate drops very quickly after a few seconds.

Mental arithmetic provides an example of how different components of working memory can be combined to solve a problem that is difficult to achieve using just one component; for example, multiply 23 by 15 without looking at this page. Information about the two numbers, and the intermediate results, has to be held in short term memory and the central executive. Now perform another multiplication, but this time look at the two numbers being multiplied (see outer margin for values), while performing the multiplication. Looking at the numbers reduces the load on working memory by removing the need to remember them.

Table 2.2 contains lists of words; those at the top of the table contain a single syllable, those at the bottom multiple syllables. Readers should have no problems remembering a sequence of five single-syllable words, a sequence of five multi-syllable words is likely to be more difficult. As before, read each word, going down a list, slowly out loud.

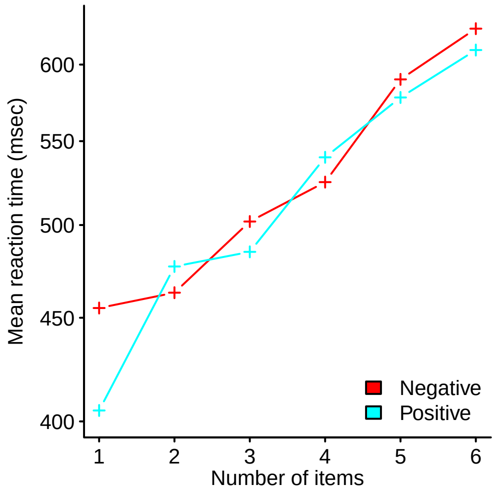


Figure 2.22: Yes/no response time (in milliseconds) as a function of number of digits held in memory. Data extracted from Sternberg.[1756] Github–Local

Making an analogy between *phonological loop* and a loop of tape in a tape recorder, suggests that it might only be possible to extract information as it goes past a *read-out point*. A study by Sternberg[1756] asked subjects to hold a sequence of digits in memory, e.g., 4185, and measured the time taken to respond yes/no, about whether a particular digit was in the sequence. Figure 2.22 shows that as the number of digits increases, the time taken for subjects to respond increases; another result was that response time was not affected by whether the answer was yes or no. It might be expected that a yes answer would enable searching to terminate, but the behavior found suggests that all digits were always being compared. Different kinds of information has different search response times.[301]

Extrapolating the results from studies based on the use of natural language,[434] to the use of computer languages, needs to take into account that reader performance has been found to

List 1	**List 2**	**List 3**	**List 4**	**List 5**
one	cat	card	harm	add
bank	lift	list	bank	mark
sit	able	inch	view	bar
kind	held	act	fact	few
look	mean	what	time	sum
ability	basically	encountered	laboratory	commitment
particular	yesterday	government	acceptable	minority
mathematical	department	financial	university	battery
categorize	satisfied	absolutely	meaningful	opportunity
inadequate	beautiful	together	carefully	accidental

Table 2.2: Words with either one or more than one syllable (and thus varying in the length of time taken to speak).

differ between words (character sequences having a recognized use in the reader's native human language) and non-words, e.g., naming latency is lower for words,[1919] and more words can be held in short term memory,[862] i.e., $word\_span = 2.4 + 2.05 \times speech\_rate$, and $nonword\_span = 0.7 + 2.27 \times speech\_rate$.

The ability to comprehend syntactically complex sentences is correlated with working memory capacity.[992] A study by Miller and Isard[1268] investigated subjects' ability to memorize sentences that varied in their degree of embedding. The following sentences have increasing amounts of embedding (figure 2.23 shows the parse-tree of two of them):

```
She liked the man that visited the jeweller that made the ring that won the prize
that was given at the fair.

The man that she liked visited the jeweller that made the ring that won the prize
that was given at the fair.

The jeweller that the man that she liked visited made the ring that won the prize
that was given at the fair.

The ring that the jeweller that the man that she liked visited made won the prize
that was given at the fair.

The prize that the ring that the jeweller that the man that she liked visited made
won was given at the fair.
```

Subjects' ability to correctly recall wording decreased as the amount of embedding increased, although performance did improve with practice. People have significant comprehension difficulties when the degree of embedding in a sentence exceeds two.[207]

Human language has a grammatical structure that enables it to be parsed serially, e.g., as it is spoken.[1462] One consequence of this expected characteristic of sentences is that so called *garden path* sentences (where one or more words at the end of a sentence changes the parsing of words read earlier) generate confusion that requires conscious effort to reason about what has been said. Examples of garden path sentences include:

```
The old train their dogs.
The patient persuaded the doctor that he was having trouble with to leave.
While Ron was sewing the sock fell on the floor.
Joe put the candy in the jar into my mouth.
The horse raced past the barn fell.
```

A study by Mathy, Chekaf and Cowan[1206] investigated the impact, on subject performance, of various patterns in a list of items to be remembered. Figure 2.24, upper plot, shows an example of how items on a list might share one or more of the attributes: color, shape or size. A list was said to be "chunkable", if its items shared attributes in a way that enabled subjects to reduce the amount of item specific information they needed to remember (e.g., all purple, small/large, triangle then circle). The items on a "non-chunkable" list did not share attributes in a way that reduced the amount of information that needed to be remembered. A chunkability value was calculated for each list.

In the simple span task subjects saw a list of items, and after a brief delay had to recall the items on the list. In the complex span task, subjects saw a list of items, and had to judge the correctness of a simple arithmetic expression, before recalling the list.

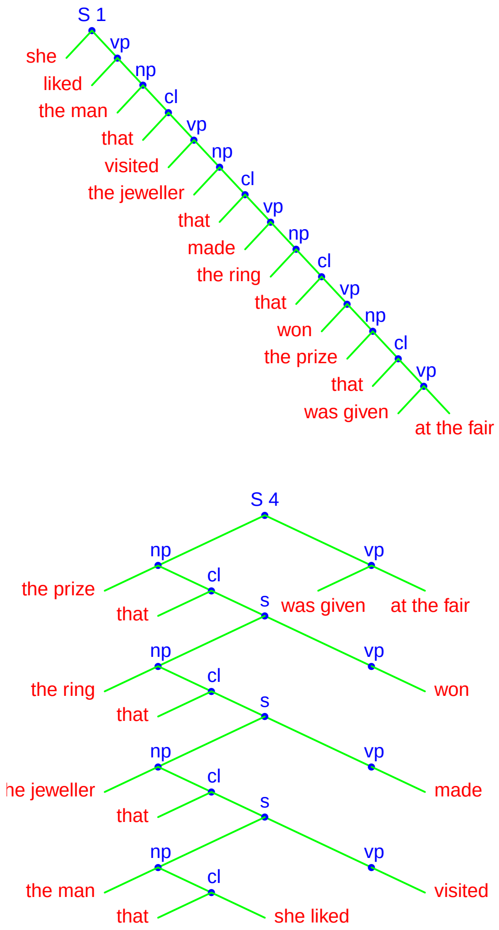


Figure 2.23: Parse tree of a sentence with no embedding, upper "S 1", and a sentence with four degrees of embedding, lower "S 4". Based on Miller et al.[1268] Github–Local

An item span score was calculated for each subject, based on the number of items correctly recalled from each list they saw (the chunkability of the list was included in the calculation). Figure 2.24, lower plot, shows the mean span over all subjects, with corresponding standard deviation.

The two competing models for loss of information in working memory are:[572] passive decay (information fades away unless a person consciously spends time rehearsing or refreshing it), and interference with new incoming information (a person has to consciously focus attention on particular information to prevent interference with new information).

2.4.2 Episodic memory

Episodic memory is memory for personally experienced events that are remembered as such, i.e., the ability to recollect specific events or episodes in our lives. When the remembered events occurred sometime ago, the term *autobiographical memory* is sometimes used.

What impact does the passage of time have on episodic memories?

A study by Altmann, Trafton and Hambrick[46] investigated the effects of interruption on a task involving seven steps. Subjects performed the same task 37 times, and were interrupted at random intervals during the 30-50 minutes it took to complete the session. Interruptions required subjects to perform a simple typing task that took, on average, 2.8, 13, 22 or 32 seconds (i.e., a very short to long interruption). Figure 2.25 shows the percentage of sequencing errors made immediately after an interruption, and under normal working conditions (in red; sequence error rate without interruptions in green). A sequence error occurs when an incorrect step is performed, e.g., step 5 is performed again, having already performed step 5, when step 6 should have been performed; the `offset` on the x-axis is the difference between the step performed, and the one that should have been performed. The sequence error rate, as a percentage of total number of tasks performed at each interruption interval, was 2.4, 3.6, 4.6 and 5.1%. The lines are predictions made by a model fitted to the data.

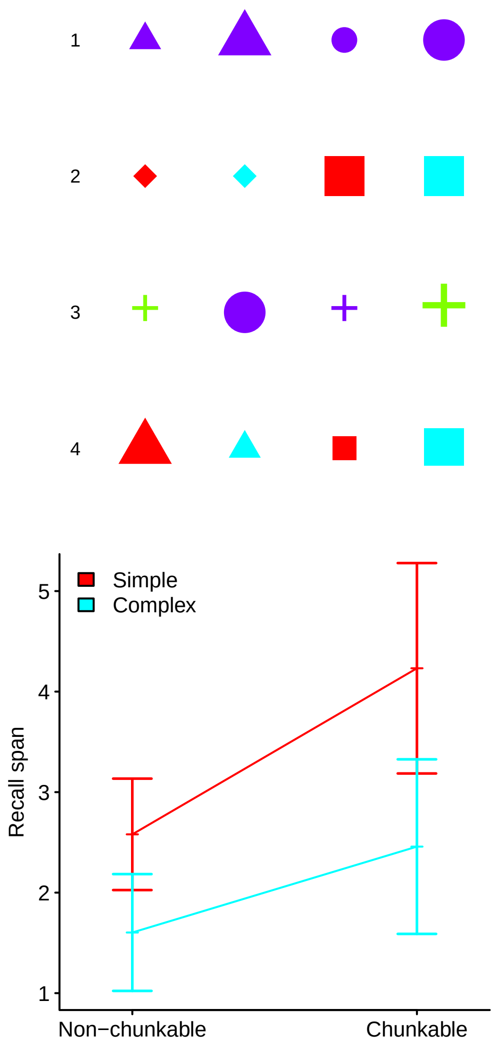


Figure 2.24: Examples of the kind of pattern of symbol sequence stimuli seen by subjects (upper); mean span over all subjects, with standard deviation (lower). Data from Mathy et al.[1206] Github–Local

2.4.3 Recognition and recall

Recognition memory is the ability to recognise previously encountered items, events or information. Studies[1024] have found that people can often make a reasonably accurate judgement about whether they know a piece of information or not, even if they are unable to recall that information at a particular instant; the so-called *feeling of knowing* is a good predictor of subsequent recall of information,

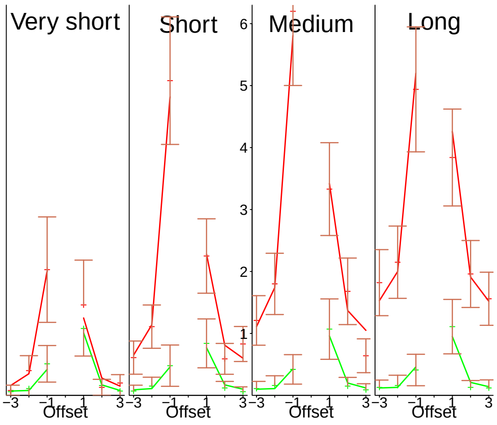


Figure 2.25: Sequencing errors (as percentage), after interruptions of various length (red), including 95% confidence intervals, sequence error rate without interruptions in green; lines are fitted model predictions. Data from Altmann et al.[46] Github–Local

Recall memory is the ability to retrieve previously encountered items, events or information. Studies have investigated factors that influence recall performance[261] (e.g., the structure of the information to be remembered, associations between new and previously learned information, and the interval between learning and recall), techniques for improving recall performance, and how information might be organized in memory.

The environment in which information was learned can have an impact on recall performance. A study by Godden and Baddeley[680] investigated subjects' recall of words memorized in two different environments. Subjects were divers, who learned a list of spoken words, either while submerged underwater wearing scuba apparatus, or while sitting at a table on dry land. The results found that subject recall performance was significantly better, when performed in the environment in which the word list was learned.

When asked to retrieve members of a category, people tend to produce a list of semantically related items, before switching to list another cluster of semantically related items and so on. This pattern of retrieval is similar to that seen in strategies of optimal food foraging,[821] however the same behavior can also emerge from a random walk on a semantic network built from human word-association data.[2]

Chunking is a technique commonly used by people to help them remember information. A chunk is a small set of items (4±1 is seen in many studies) having a common, strong, association with each other (and a much weaker one to items in other chunks). For instance, Wickelgren[1933] found that people's recall of telephone numbers is optimal, if numbers are grouped into chunks of three digits. An example, using random-letter sequences is: **fbicbsibmirs**. The trigrams (**fbi**, **cbs**, **ibm**, **irs**), within this sequence of 12 letters, are well-known acronyms in the U.S.A. A person who notices this association can

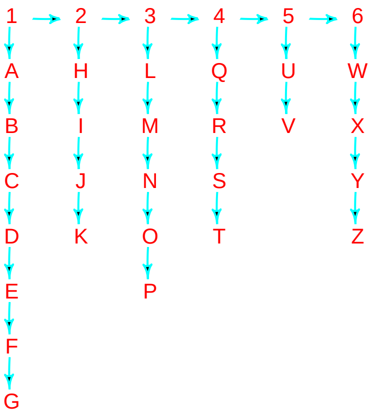


Figure 2.26: Semantic memory representation of alphabetic letters (the numbers listed along the top are place markers and are not stored in subject memory). Readers may recognize the structure of a nursery rhyme in the letter sequences. Derived from Klahr.[1005] Github–Local

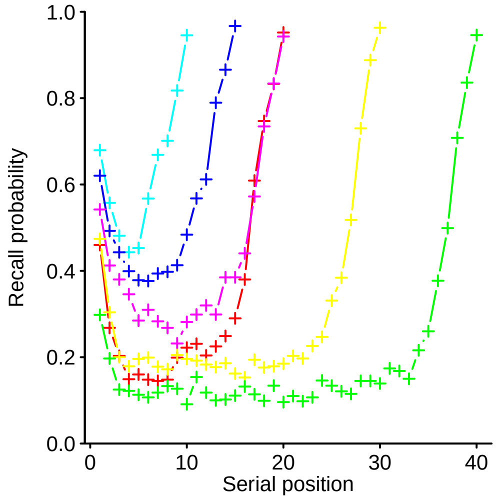


Figure 2.27: Probability of correct recall of words, by serial presentation order; for lists of length 10, 15 and 20 each word visible for 1, for lists of length 20, 30 and 40 each word visible for 2 seconds. Data from Murdoch,[1313] via Brown.[261] Github–Local

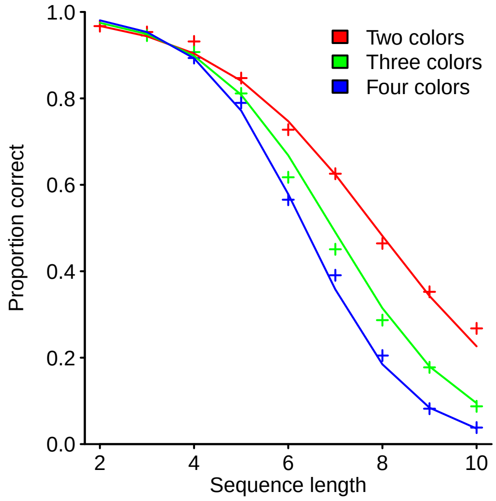


Figure 2.28: Proportion of correctly recalled colored dot sequences of a given length, containing a given number of colors; lines are fitted regression models. Data kindly provided by Chekaf.[333] Github–Local

use it to aid recall. Several theoretical analyses of memory organizations have shown that chunking of items improves search efficiency (optimal chunk size 3–4,[495] number items at which chunking becomes more efficient than a single list, 5–7[1170]).

A study by Klahr, Chase, and Lovelace[1005] investigated how subjects stored letters of the alphabet in memory. Through a series of time-to-respond measurements, where subjects were asked to name the letter that appeared immediately before or after the presented probe letter, they proposed the alphabet-storage structure shown in figure 2.26.

2.4.3.1 Serial order information

Information about the world is processed serially. Studies[869] have consistently found a variety of patterns in recall of serial information (studies often involve recalling items from a recently remembered list), patterns regularly found include:

- higher probability of recall for items at the start (the *primacy effect*) and end (the *recency effect*) of a list (known as the *serial position effect*;[1313] see figure 2.27). The probability that an item will be remembered as occupying position i, at time $t+1$, is approximately given by,[1331] for interior positions: $P_{i,t+1} = (1-\theta)P_{i,t} + \frac{\theta}{2}P_{i-1,t} + \frac{\theta}{2}P_{i+1,t}$, and for the first item: $P_{1,t+1} = (1-\frac{\theta}{2})P_{1,t} + \frac{\theta}{2}P_{2,t}$, and similarly for the last time. One study,[1331] involving lists of five words, found that using $\theta = 0.12$, for each 2-hour retention interval, produced predictions in reasonable agreement with the experimental data (more accurate models have been created[261]),
- recall of a short list tends to start with the first item, and progress in order through the list, while for a long list people are more likely to start with one of the last four items.[1906] When prompted by an entry on the list people are most likely to recall the item following it;[852] see Github–developers/misc/HKJEP99.R,
- when recalling items from an ordered list, people tend to make anticipation errors (i.e., recall a later item early), shifting displaced items further along the list.[571]

A method's parameters have a serial order, and the same type may appear multiple times within the parameter list.

A study by Chekaf, Gauvrit, Guida, and Mathy[333] investigated subjects' recall performance of a sequence of similar items. Subjects saw a sequence of colored dots, each dot visible for less than a second, and had to recall the presented sequence of colors. The number of colors present in a sequence varied from two to four, and the sequence length varied from two to ten.

Figure 2.28 shows the proportion of correctly recalled dot sequences of a given length, containing a given number of colors; lines are fitted Beta regression models.

A study by Adelson[9] investigated the organization present in subject's recall order of previously remembered lines of code. Subjects saw a total of 16 lines of code, one line at a time, and were asked to memorise each line for later recall. The lines were taken from three simple programs (two containing five lines each and one containing six lines), the program order being randomised for each subject. Five subjects were students who had recently taken a course involving the language used, and five subjects had recently taught a course using the language.

Subjects were free to recall the previously seen lines in any order. An analysis of recall order showed that students grouped lines by syntactic construct (e.g., loop, function header), while the course teachers recalled the lines in the order in which they appeared in the three programs (i.e., they reconstructed three programs from the 16 randomly ordered lines); see figure 2.29.

A study by Pennington[1447] found that developers responded slightly faster to questions about a source code statement when its immediately preceding statement made use of closely related variables.

2.4.4 Forgetting

People are unhappy when they forget things; however, not forgetting may be a source of unhappiness.[1367] The Russian mnemonist Shereshevskii found that his ability to remember everything cluttered up his mind.[1162] Having many similar, not recently used, pieces of information matching during a memory search can be counterproductive; forgetting is a useful adaptation.[1630] For instance, a driver returning to a car wants to know where it was last parked, not the location of all previous parking locations. It has been proposed that human memory is optimized for information retrieval based on the statistical properties of the likely need for the information,[57] in peoples' everyday lives (which, these days, includes the pattern of book borrowings from libraries[277]). The rate at which the mind forgets seems to mirror the way that information tends to lose its utility, over time, in the world in which we live. Organizational forgetting is discussed in section 3.4.5.

Some studies of forgetting have found that a power law is a good fit to the reduction, over time, in subjects' ability to correctly recall information,[1595] while results from other studies are better fitted by an exponential equation (over the measurement range of many experiments, the difference between the two fitted models is usually small).[v] As more experimental data has become available, more complicated models have been proposed.[1241]

A study by Ebbinghaus,[518] using himself as a subject, performed what has become a classic experiment in forgetting. The measure of learning and forgetting used was based on the relative time saved, when relearning previously learned information, compared to the time taken to first learn the information. Ebbinghaus learned lists of 104 nonsense syllables, with intervals between relearning of 20 minutes, 1 hour, 9 hours, 1 day, 2 days, 6 days and 31 days. Figure 2.30 shows the fraction of time saved, after a given duration, when relearning list contents to the original learned standard (with standard errors)

Another measure of information retention was used in a study by Rubin, Hinton and Wenzel.[1594] Subjects saw a pair of words on a screen, which they had to remember; later, when the first word appeared on the screen, they had to type the second word of the pair. Subjects saw a sequence of different word pairs; *lag* is defined as the number of words seen between first seeing a word pair and being asked to give the second word in response to the appearance of the first word of the pair.

Figure 2.31 shows the fraction of correct second words at various lag intervals. The red line is a fitted bi-exponential regression model, with blue and green lines showing its two exponential components.

A study by Meeter, Murre and Janssen[1241] investigated the likelihood of prominent of news stories being correctly recalled over a period of 16 months. The questions subjects were asked had two possible forms: forced choice, where four possible answers were listed and one had to be selected, and: open, where an answer had to be provided with no suggestions listed. Data was collected from 14,000 people, via an internet news test. Figure 2.32 shows the fraction of correct answers each day, against time elapsed since the event in the question was reported.

2.5 Learning and experience

Humans are characterized by an extreme dependence on culturally transmitted information.

People have the ability to learn, and on many tasks human performance improves with practice, e.g., time taken to deciding whether a character sequence is an English word.[979] Many studies have fitted a power law to measurements of practice performance (the term *power law of learning* is often used). If chunking is assumed to play a role in learning, a power law is a natural consequence;[1350] the equation has the form:[vi]

$RT = a + bN^{-c}$, where: RT is the response time, N the number of times the task has been performed, and a, b, and c are constants obtained by model fitting.

There are also theoretical reasons for expecting the measurements to be fitted by an exponential equation, and this form of model has been fitted to many learning data sets;[790] the equation has the form:[vii]

$$RT = a + be^{-cN}$$

Both equations often provide good enough fits to the available data, and without more measurements than is usually available, it is not possible to show one is obviously better than the other.

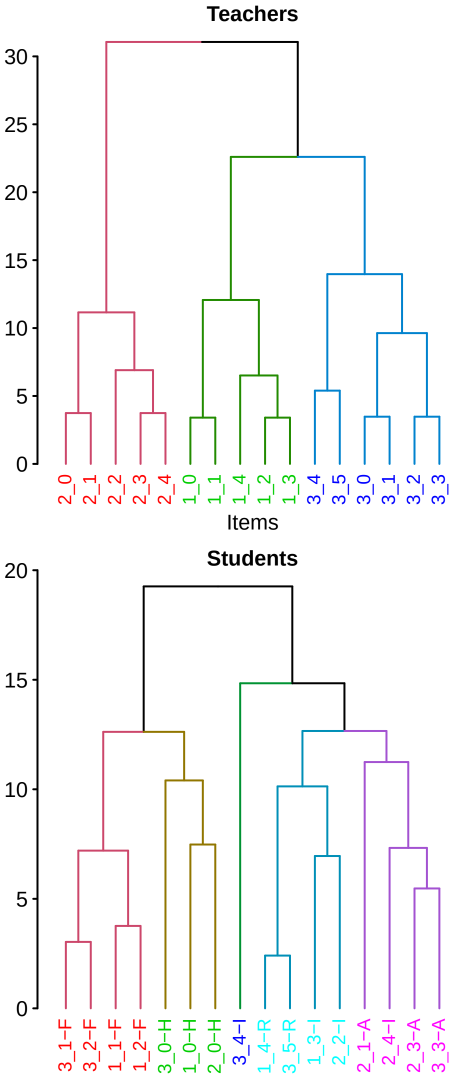


Figure 2.29: Hierarchical clustering of statement recall order, averaged over teachers and students; label names are: program_list-statementkind, where statementkind might be a function header, loop, etc. Data extracted from Adelson.[9] Github–Local

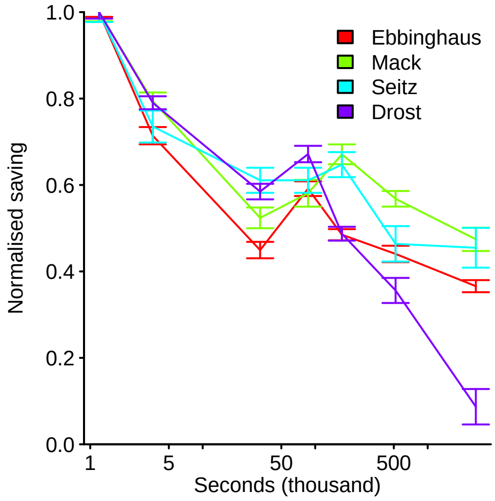


Figure 2.30: Fraction of relearning time saved (normalised) after given interval since original learning; original Ebbinghaus study and three replications (with standard errors). Data from Murre et al.[1317] Github–Local

[v] It has been suggested that the power laws are a consequence of fitting data averaged over multiple subjects; see section 9.3.3.

[vi] Power laws can be a consequence of fitting data averaged over multiple subjects, rather than representing individual subject performance; see section 9.3.3.

[vii] Similar equations can also be obtained by averaging over a group of individuals whose learning takes the form of a step function.[638]

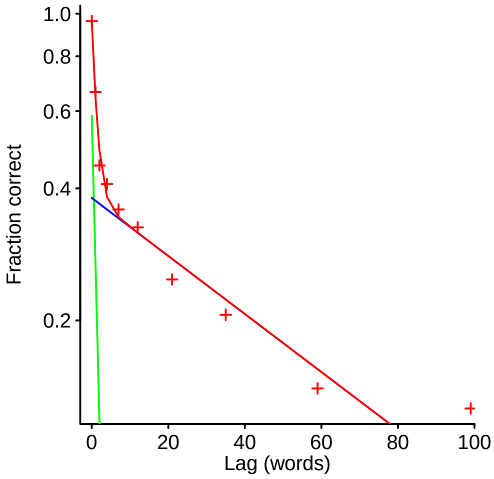


Figure 2.31: Fraction of correct subject responses, with fitted bi-exponential model in red (blue and green lines are its two exponential components). Data from Rubin et al.[1594] Github–Local

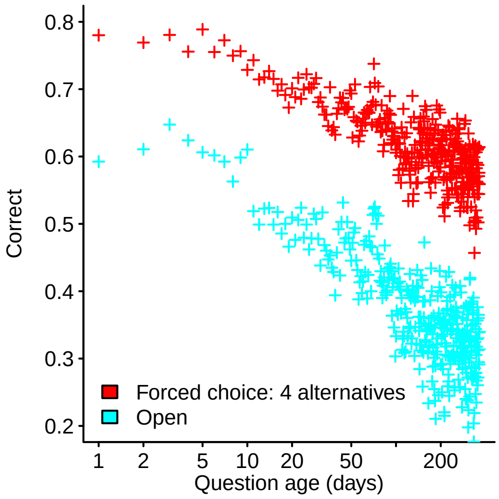


Figure 2.32: Fraction of news items correctly recalled each day, after a given number of days since the event; Forced choice of one alternative from four, and Open requiring an answer with no suggestions provided. Data from Meeter et al.[1241] Github–Local

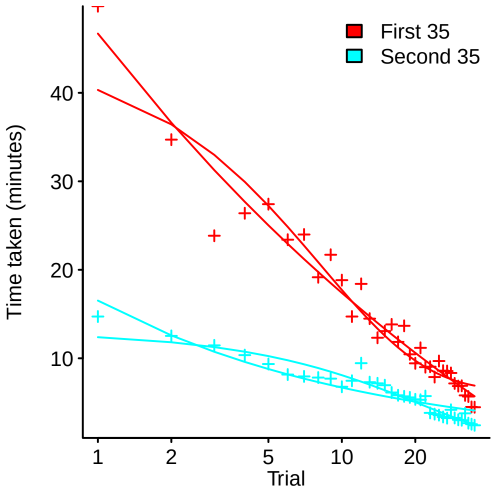


Figure 2.33: Time taken to solve the same jig-saw puzzle 35 times, followed by a two-week interval and then another 35 times, with power law and exponential fits. Data extracted from Alteneder.[43] Github–Local

Implicit learning occurs when people perform a task containing information that is not explicitly obvious to those performing it. A study by Reber and Kassin[1546] compared implicit and explicit pattern detection. Subjects were asked to memorize sets of words, with the words in some sets containing letter sequences generated using a finite state grammar. One group of subjects thought they were just taking part in a purely memory-based experiment, while the second group were told of the existence of a letter sequence pattern in some words, and that it would help their performance if they could deduce this pattern. The performance of the two groups, on the different sets of words (i.e., pattern words only, pattern plus non-pattern words, non-pattern words only), matched each other. Without being told to do so, subjects had used patterns in the words to help perform the memorization task.

Explicit learning occurs when the task contains patterns that are apparent, and can be remembered and used on subsequent performances. A study by Alteneder[43] recorded the time taken, by the author, to solve the same jig-saw puzzle 35 times (over a four-day period). After two weeks, the same puzzle was again solved 35 times. Figure 2.33 shows both a fitted power law and exponential; the exponent of the fitted power law, for the first series, is -0.5

Social learning, learning from others, is discussed in section 3.4.4, and organizational learning is discussed in section 3.4.5.

For simple problems, learning can result in the solution being committed to memory; performance is then driven by reaction-time, i.e., the time needed to recall and give the response. Logan[1144] provides an analysis of subject performance on these kinds of problems.

The amount of practice needed to learn any patterns present in a task (to be able to perform it), depends on the complexity of the patterns. A study by Kruschke[1034] asked subjects to learn the association between a stimulus and a category; they were told that the category was based on height and/or position, During the experiment subjects were told whether they had selected the correct category for the stimulus. When the category involved a single attribute (i.e., one of height or location), learning was predicted to be faster, compared to when it involved two attributes, i.e., learning difficulty depends on the number of variables and states.

Figure 2.34 shows the probability of a correct answer, for a particular kind of stimulus (only height or position, and some combination of height and position), for eight successive blocks of eight stimulus/answer responses.

The patterns present in a task may change. What impact do changes in task characteristics have on performance? A study by Kruschke[1035] asked subjects to learn the association between a stimulus and a category described by three characteristics (which might be visualized in a 3-D space; see fig 2.46). Once their performance was established at close to zero errors, the characteristics of the category were changed. The change pattern was drawn from four possibilities: reversing the value of each characteristic (considered to be a zero-dimension change), changes to one characteristic, two characteristics and three characteristics (not reversal).

Figure 2.35 shows average subject performance improving to near perfect (over 22 successive blocks of eight stimulus/answer responses), a large performance drop when the category changes, followed by improving performance, over successive blocks, as subjects learn the new category. The change made to the pattern for the learned category has a noticeable impact on the rate at which the new category is learned.

The source code for an application does not have to be rewritten every time somebody wants a new copy of the program; it is rare for a developer to be asked to reimplement exactly the same application again. However, having the same developer reimplement the same application, multiple times, provides information about the reduced implementation time that occurs with practice.

A study by Lui and Chan[1156] asked 24 developers to implement the same application four times; 16 developers worked in pairs (i.e., eight pair programming teams) and eight worked solo. Before starting to code, the subjects took a test involving 50 questions from a computer aptitude test; subjects were ranked by number of correct answers, and pairs selected such that both members were adjacent in the ranking.

Learning occurs every-time the application is implemented, and forgetting occurs during the period between implementations (each implementation occurred on a separate weekend, with subjects doing other work during the week) . Each subject had existing knowledge and skill, which means everybody started the experiment at a different point on the learning curve. In the following analysis the test score is used as a proxy for each subject's initial point on the learning curve.

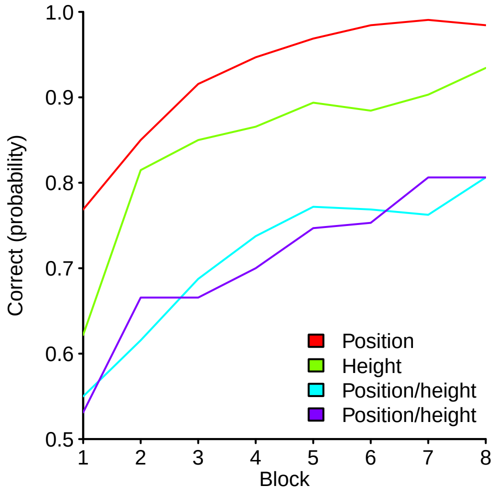


Figure 2.34: Probability of assigning a stimulus to the correct category, where the category involved: height, position, and a combination of both height and position. Data from Kruschke.[1034] Github–Local

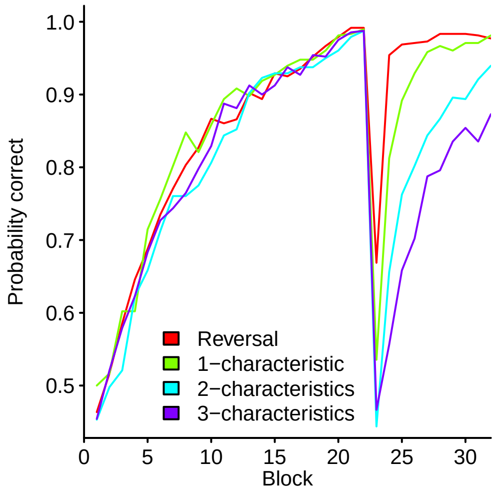


Figure 2.35: Probability of assigning a stimulus to the correct category; learning the category, followed in block 23 by a change in the characteristics of the learned category. Data from Kruschke.[1035] Github–Local

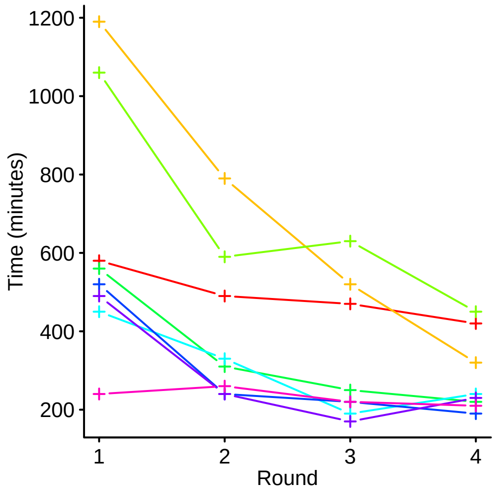


Figure 2.36: Completion times of eight solo developers for each implementation round. Data kindly provided by Lui.[1156] Github–Local

Figure 2.36 shows the completion times, for each round of implementation, for solo and pairs. The equation: $Completion\_time = a \times (b \times Test\_score + Round)^c$, provides a good fit, where: $Completion\_time$ is time to complete an implementation of the application, $Test\_score$ the test score and $Round$ the number of times the application has been implemented, with a, b and c constants chosen by the model fitting process. However, its predictions are in poor agreement with actual values, suggesting that other factors are making a significant contribution to performance.

After starting work in a new environment, performance can noticeable improve as a person gains experience working in that environment. A study by Brooks[259] measured the performance of an experienced developer, writing and debugging 23 short programs (mean length 24 lines). The time taken to debug each program improved as more programs were implemented; see Github–developers/a013582.R.

Developers sometimes work in several programming languages on a regular basis. The extent to which learning applies across languages is likely to be dependent on the ease with which patterns of usage are applicable across the languages used.

A study by Zislis[2010] measured the time taken (in minutes) by himself, to implement 12 algorithms, with the implementation performed three times using three different languages (APL, PL/1, Fortran), and on the fourth repetition using the same language as on the first implementation. Figure 2.37 shows total implementation time for each algorithm, over the four implementations; fitting a mixed-effects model finds some performance difference between the languages used (see figure code for details).

A study by Jones[920] investigated developer beliefs about binary operator precedence. Subjects saw an expression containing two binary operators, and had to specify the relative precedence of these operators by adding parenthesis to the expression, e.g., `a + b | c`. In a coding environment, the more frequently a pair of binary operators appear together, the more often developers have to make a decision about their relative precedence; the hypothesis was that the more often developers have to make a particular precedence decision, when reading code, the more likely they are to know the correct answer. Binary operator usage in code was used as a proxy for developer experience of making binary operator decisions (C source code was measured). Figure 2.38 shows the fraction of correct answers to the relative operator precedence question, against the corresponding percentage occurrence of that pair of binary operators.

A study by Mockus and Weiss[1286] found that the probability of a developer introducing a fault into an application, when modifying software, decreased as the log of the total number of changes made by the developer, i.e., their experience or expertise.

Job advertisements often specify that a minimum number of years of experience is required. Number of years may not be a reliable measure of expertise, but it does provide a degree of comfort that a person has had to deal with the major issues that might occur within a given domain.

A study by Latorre[1080] investigated the effect of developer experience on applying unit-test-driven development. The 24 subjects, classified as junior (one-to-two-years professional experience), intermediate (three-to-five-years experience) or senior (more than six-years experience), were taught unit-test-driven development, and the time taken for them to implement eight groups of requirements was measured. The implementations of the first two groups was included as part of the training and familiarization process; the time taken, by each subject, on these two groups was used to normalise the reported results.

Figure 2.39 shows the normalised time taken, by each subject, on successive groups of requirements; color is used to denote subject experience. While there is a lot of variation between subjects, average performance improves with years of experience, i.e., implementation time decreases (a fitted mixed-effects model is included in the plot's code).

What is the long term impact of learning on closely related cognitive activities? Isaac Asimov was a prolific author who maintained a consistent writing schedule. Figure 2.40 shows the number of published books against elapsed months. The lines are the fitted models $books \approx 27 months^{0.48}$, and $books \approx -0.6 months + 40\sqrt{months}$ (a better fit).

To quote Herbert Simon:[1690] "Intuition and judgement–at least good judgement–are simply analyses frozen into habit and into the capacity for rapid response through recognition. . . . Every manager needs also to be able to respond to situations rapidly, a skill that requires the cultivation of intuition and judgement over many years of experience and training.”

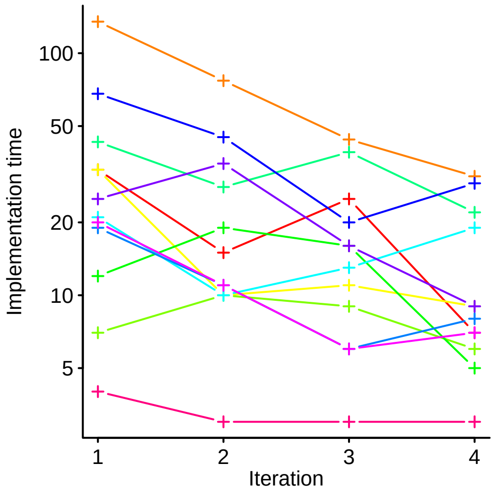


Figure 2.37: Time taken, by the same person, to implement 12 algorithms from the Communications of the ACM (each colored line), with four iteration of the implementation process. Data from Zislis.[2010] Github–Local

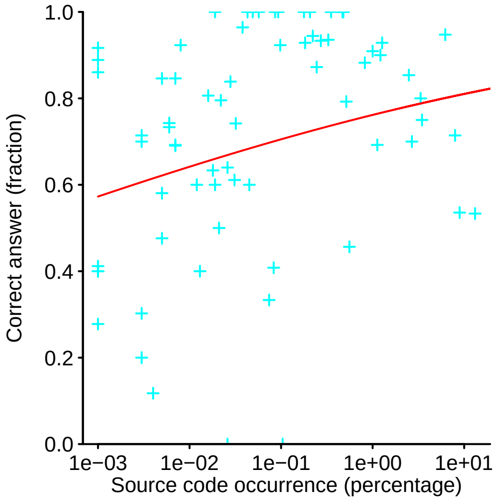


Figure 2.38: Percentage occurrence of binary operator pairs (as a percentage of all such pairs) against the fraction of correct answers to questions about their precedence, red line is beta regression model. Data from Jones.[920] Github–Local

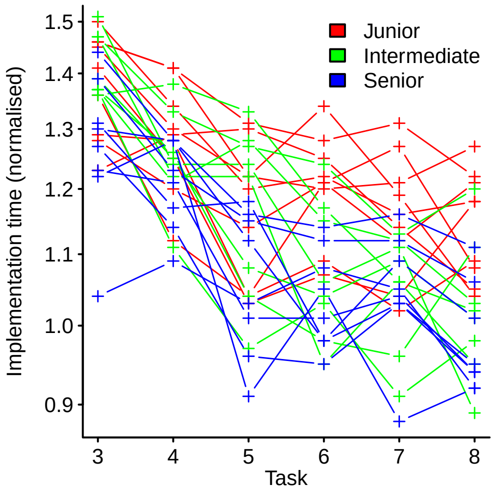


Figure 2.39: Time taken by 24 subjects, classified by years of professional experience, to complete successive tasks. Data from Latorre.[1080] Github–Local

2.5.1 Belief

People hold beliefs derived from the information they have received, and the analysis of information accumulated over time.

How are existing beliefs modified by the introduction of new evidence?

In the belief-adjustment model[834] the current degree of belief has the form of a moving average of updates produced by the history of prior encounters with items of evidence:

$S_k = S_{k-1} + w_k[s(x_k) - R]$, where: $0 < S_k < 1$ is the degree of belief in some hypothesis after evaluating k items of evidence, S_{k-1} is the prior belief (S_0 denotes the initial belief), $s(x_k)$ is the subjective evaluation of the kth item of evidence (different people may assign different values for the same evidence, x_k), R is the reference point or background, against which the impact of the kth item of evidence is evaluated, and $0 < w_k < 1$ is the adjustment weight for the kth item of evidence.

When presented with an item of evidence, a person can use an evaluation process or an estimation process.

- an evaluation process encodes new evidence relative to a fixed point, i.e., the hypothesis addressed by a belief. If the new evidence supports the hypothesis, a person’s belief increases, and decreases if it does not support the hypothesis. This change occurs irrespective of the current state of a person’s belief; for this case $R = 0$, and $-1 \leq s(x_k) \leq 1$.

  An example of an evaluation process might be the belief that the object X always holds a value that is numerically greater than Y,
- an estimation process encodes new evidence relative to the current state of a person’s beliefs; for this case $R = S_{k-1}$, and $0 \leq s(x_k) \leq 1$.

If multiple items of evidence are presented, a response may be required after each item (known as *Step-by-Step*), or a response may only need to be given after all the evidence is presented (known as *End-of-Sequence*). The response mode is handled by the model.

A study by Hogarth and Einhorn[834] investigated order, and response mode effects in belief updating. Subjects were presented with a variety of scenarios (e.g., a defective stereo speaker thought to have a bad connection, or a baseball player whose hitting improved dramatically after a new coaching program), followed by two or more additional items of evidence. The additional evidence was either positive (e.g., “The other players on Sandy’s team did not show an unusual increase in their batting average over the last five weeks”), or negative (e.g., “The games in which Sandy showed his improvement were played against the last-place team in the league”). The positive and negative evidence was worded to create either strong or weak forms.

The evidence was presented in three combinations: strong positive and then weak positive, upper plot in figure 2.41; strong negative and then weak negative, middle plot of figure 2.41; positive negative and then negative positive, lower plot of figure 2.41. Subjects were then asked, “Now, how likely do you think X caused Y on a scale of 0 to 100?” For some presentations, subjects had to respond after seeing each item of evidence (the step-by-step procedure), in the other presentations subjects did not respond until seeing all the evidence (the end-of-sequence procedure).

Figure 2.41 shows the impact of presentation orders and response modes on subjects’ degree of belief.

Other studies have replicated these results, for instance, professional auditors have been shown to display recency effects in their evaluation of the veracity of company accounts.[1431]

One study[538] found that when combining information from multiple sources, to form beliefs, subjects failed to adjust for correlation between the information provided by different sources (e.g., news websites telling slightly different versions of the same events). Failing to adjust for correlation results in the creation of biased beliefs.

Studies[1584] have found that people exhibit a belief preservation effect; they continue to hold beliefs after the original basis for those beliefs no longer holds.

The mathematical approach used to model quantum mechanics is started being used to model some cognitive processes, such as order effects and holding conflicting beliefs at the same time.[1826]

2.5.2 Expertise

The term *expert* might be applied to a person because of what other people think (i.e., be socially based), such as professional standing within an organization[viii], and self-proclaimed experts willing to accept money from clients who are not willing to take responsibility for proposing what needs to be done[69] (e.g., the role of court jester who has permission to say what others cannot); or, be applied to a person because of what they can do (i.e., performance based), such as knowing a great deal about a particular subject, or being able to perform at a qualitatively higher level than the average person, or some chosen top percentage, or be applied to a person having a combination of social and performance skills.[679]

This section discusses expertise as a high-performance skill; something that requires many years of training, and where many individuals fail to develop proficiency.

How might people become performance experts?

Chess players were the subject of the first major study of expertise, by de Groot,[443] and techniques used to study Chess, along with the results obtained, continue to dominate the study of expertise. In a classic study, de Groot briefly showed subjects the position of an unknown game and asked them to reconstruct it. The accuracy and speed of experts (e.g., Grand Masters) was significantly greater than non-experts when the pieces appeared on the board in positions corresponding to a game, but was not much greater when the pieces were placed at random. The explanation given for the significant performance difference, is that experts are faster to recognise relationships between the positions of pieces, and make use of their large knowledge of positional patterns to reduce the amount of working memory needed to remember what they were briefly shown.

A study by McKeithen, Reitman, Ruster and Hirtle[1228] gave subjects two minutes to study the source code of a program, and then gave them three minutes to recall the program's 31 lines. Subjects were then given another two minutes to study the same source code, and asked to recall the code; this process was repeated for a total of five trials.

Figure 2.42 shows the number of lines recalled by experts (over 2,000 hours of general programming experience), intermediates (just completed a programming course), and beginners (about to start a programming course) over the five trials. The upper plot are the results for the 31 line program, and the lower plot a scrambled version of the program.

Some believe that experts have an innate ability, or capacity, that enables them to do what they do so well. Research has shown that while innate ability can be a factor in performance (there do appear to be genetic factors associated with some athletic performances), the main factor in developing expert performance is time spent in *deliberate practice*[543] (deliberate practice does not explain everything[765]).

Deliberate practice differs from simply performing the task,[544] in that it requires people to monitor their practice with full concentration, and to receive feedback[835] on what they are doing (often from a professional teacher). The goal of deliberate practice being to improve performance, not to produce a finished product, and may involve studying components of the skill in isolation attempting to improve on particular aspects. The goal of this practice being to improve performance, not to produce a finished product.

A study by Lorko, Servátka and Zhang[1148] investigated the effect of lack of feedback, and anchoring, on the accuracy of duration estimates of a repeatedly performed task. One round of the task involved making an estimate of time to complete the task, followed by giving 400 correct true/false answers to questions involving an inequality between two numbers (whose value was between 10 and 99); there were three rounds of estimating and correctly answering 400 questions. The amount of money paid to subjects, for taking part, was calculated using: $earnings = 4.5 - 0.05(\text{abs}(actual - estimate))$, and a performance formula involving number of correct and incorrect answers given; the intent was to motivate subjects to provide an accurate estimate, and to work as quickly as possible without making mistakes. Subjects did not receive any feedback on the accuracy of their estimates, i.e., they were not told how much time they took to complete the task.

The task was expected to take around 10 to 12.5 minutes. Approximately one third of subjects were given a low anchor (i.e., 3-minutes), one third high a high anchor (i.e., 20-minutes), and the other third no anchor.

The results show that the estimates of low-anchor subjects increased with each round, while high-anchor subject estimates decreased, and no anchor subject estimates showed no pattern; see Github–developers/Lorko-Servatka-Zhang.R.

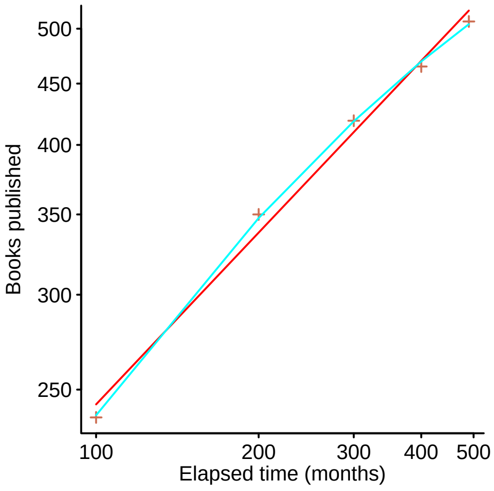


Figure 2.40: Elapsed months during which Asimov published a given number of books, with lines for two fitted regression models. Data from Ohlsson.[1389] Github–Local

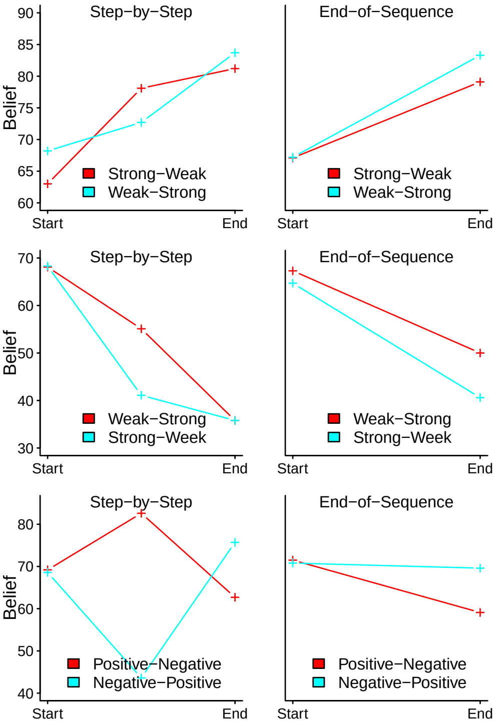


Figure 2.41: Subjects' belief response curves when presented with evidence in the sequences: (upper) positive weak, then positive strong, (middle) negative weak then negative strong, (lower) positive then negative. Based on Hogarth et al.[834] Github–Local

[viii] Industrial countries use professionalism as a way of institutionalising expertise.

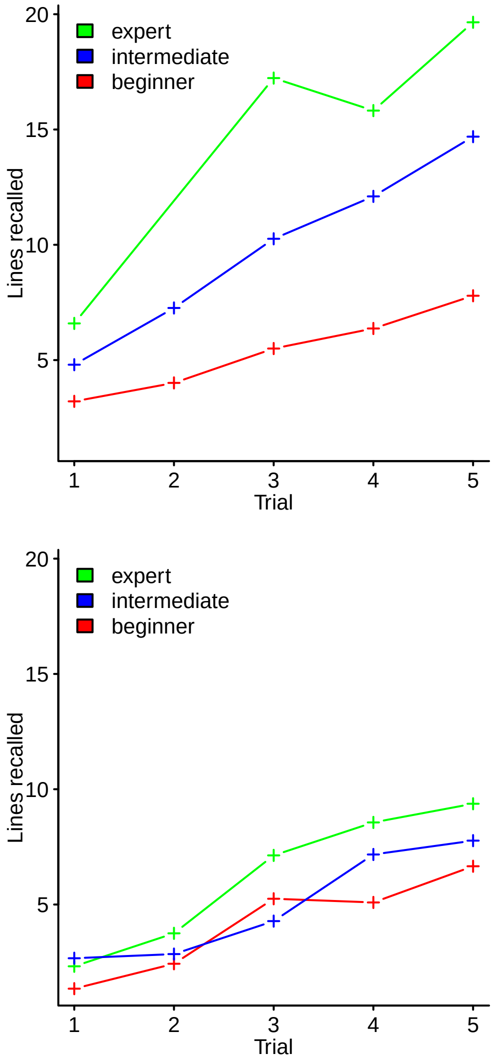


Figure 2.42: Lines of code correctly recalled after a given number of 2-minute memorization sessions; actual program in upper plot, scrambled line order in lower plot. Data extracted from McKeithen et al.[1228] Github–Local

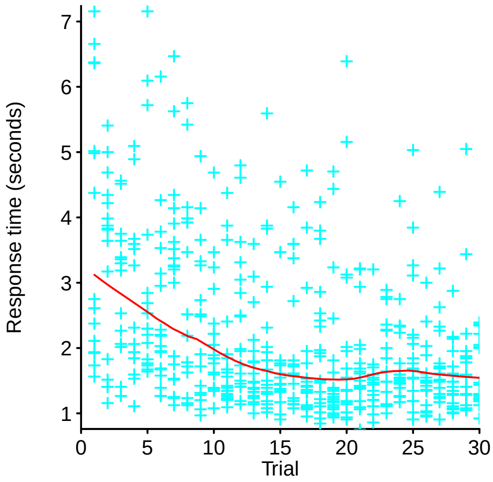


Figure 2.43: One subject's response time over successive blocks of command line trials and fitted loess (in green). Data kindly provided by Remington.[1554] Github–Local

In many fields expertise is acquired by memorizing a huge amount of domain-specific information, organizing it for rapid retrieval based on patterns that occur when problem-solving within the domain, and refining the problem-solving process.[546]

Studies of the backgrounds of recognized experts, in many fields, found that the elapsed time between them starting out and carrying out their best work was at least 10 years, often with several hours of deliberate practice every day of the year. For instance, a study of violinists[545] (a perceptual-motor task) found that by age 20 those at the top-level had practiced for 10,000 hours, those at the next level down 7,500 hours, and those at the lowest level of expertise had practiced for 5,000 hours; similar quantities of practice were found in those attaining expert performance levels in purely mental activities (e.g., chess).

Expertise within one domain does not confer any additional skills within another domain,[55] e.g., statistics (unless the problem explicitly involves statistical thinking within the applicable domain), and logic.[338] A study[330] in which subjects learned to remember long sequences of digits (after 50–100 hours of practice they could commit to memory, and later recall, sequences containing more than 20 digits) found that this expertise did not transfer to learning sequences of other items.

There are domains in which those acknowledged as experts do not perform significantly better than those considered to be non-experts,[286] in some cases non-experts have been found to outperform experts within their domain.[1939] An expert's domain knowledge can act as a mental set that limits the search for a solution, with the expert becoming fixated within the domain. In cases where a new task does not fit the pattern of highly proceduralized behaviors of an expert, a novice has an opportunity to do better.

What of individual aptitudes? In the cases studied, the effects of aptitude, if there was any, were found to be completely overshadowed by differences in experience, and deliberate practice times. Willingness to spend many hours, every day, studying to achieve expert performance is certainly a necessary requirement. Does an initial aptitude, or interest, in a subject lead to praise from others (the path to musical and chess expert performance often starts in childhood), which creates the atmosphere for learning, or are other issues involved? IQ scores do correlate to performance during, and immediately after training, but the correlation reduces over the years. The IQ scores of experts has been found to be higher than the average population, at about the level of college students.

Education can be thought of as trying to do two things (of interest to us here)—teach students skills (procedural knowledge), and providing them with information, considered important in the relevant field, to memorize (declarative knowledge).

Does attending a course on a particular subject have any measurable effect on students' capabilities, other than being able to answer questions in an exam? That is, having developed some skill in using a particular system of reasoning, do students apply it outside the domain in which they learnt it?

A study by Lehman, Lempert, and Nisbett[1099] measured changes in students' statistical, methodological and conditional reasoning abilities (about everyday-life events) between their first and third years. They found that both psychology and medical training produced large effects on statistical and methodological reasoning, while psychology, medical and law training produced effects on the ability to perform conditional reasoning; training in chemistry had no effect on the types of reasoning studied. An examination of the skills taught to students studying in these fields showed that they correlated with improvements in the specific types of reasoning abilities measured.

A study by Remington, Yuen and Pashler[1554] compared subject performance between using a GUI and a command line (with practice, there was little improvement in GUI performance, but command line performance continued to improve and eventually overtook GUI performance). Figure 2.43 shows the command line response time for one subject over successive blocks of trials, and a fitted loess line.

2.5.3 Category knowledge

Children as young as four have been found to use categorization to direct the inferences they make,[658] and many studies have shown that people have an innate desire to create and use categories (they have also been found to be sensitive to the costs and benefits of using categories;[1177] Capuchin monkeys have learned to classify nine items concurrently[1226]). By dividing items into categories, people reduce the amount of information they need to learn,[1494] and can generalize based on prior experience.[1365] Information about the likely characteristics of a newly encountered item can be obtained by matching it to one or more known categories, and then extracting characteristics common to previously encountered items in these categories. For instance, a flying object with feathers, and a beak might be assigned to the category *bird*, which suggests the characteristics of laying eggs and potentially being migratory.

Categorization is used to perform inductive reasoning (i.e., the derivation of generalized knowledge from specific instances), and also acts as a memory aid (about the members of a category). Categories provide a framework from which small amounts of information can be used to infer, seemingly unconnected (to an outsider), useful conclusions.

Studies have found that people use roughly three levels of abstraction in creating hierarchical relationships. The highest level of abstraction has been called[1579] the *superordinate-level* (e.g., the general category furniture), the next level down the *basic-level* (this is the level at which most categorization is carried out, e.g., car, truck, chair or table), the lowest level is the *subordinate-level* (which denotes specific types of objects, e.g., a family car, a removal truck, my favourite armchair, a kitchen table). Studies[1579] have found that basic-level categories have properties not shared by the other two categories, e.g., adults spontaneously name objects at this level, it is the abstract level that children acquire first, and category members tend to have similar overall shapes.

When categories have a hierarchical structure, it is possible for an attribute of a higher-level category to affect the perceived attributes of subordinate categories. A study by Stevens and Coupe[1757] asked subjects to remember the information contained in a series of maps (see figure 2.44). They were asked questions such as: "Is X east or west of Y?", and "Is X north or south of Y?" Subjects gave incorrect answers 18% of the time for the congruent maps (i.e., country boundary aligned along axis of question asked), but 45% of the time for the incongruent maps (i.e, country boundary meanders, and locations sometimes inconsistent with question asked); 15% for homogeneous (i.e., no country boundaries). These results were interpreted as subjects using information about the relative locations of countries to answer questions about the city locations.

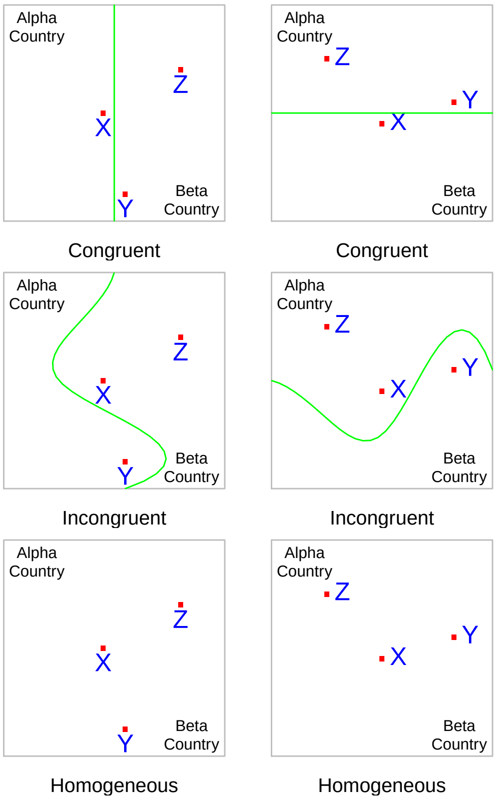


Figure 2.44: Country boundaries (green line) and town locations (red dots). Congruent: straight boundary aligned with question asked, incongruent: meandering boundary and locations sometimes inconsistent with question asked. Based on Stevens et al.[1757] Github–Local

Studies[1703] have found that people do not consistently treat subordinate categories as inheriting the properties of their superordinates, i.e., category inheritance is not always a tree.

How categories should be defined and structured is a long-standing debate within the sciences. Some commonly used category formation techniques, their membership rules and attributes include:

- defining-attribute theory: members of a category are characterized by a set of *defining attributes*. Attributes divide objects up into different concepts whose boundaries are well-defined, with all members of the concept being equally representative. Concepts that are a basic-level of a superordinate-level concept will have all the attributes of that superordinate level; for instance, a sparrow (small, brown) and its superordinate: bird (two legs, feathered, lays eggs),
- prototype theory: categories have a central description, the *prototype*, that represents the set of attributes of the category. This set of attributes need not be necessary, or sufficient, to determine category membership. The members of a category can be arranged in a typicality gradient, representing the degree to which they represent a typical member of that category. It is possible for objects to be members of more than one category, e.g., tomatoes as a fruit, or a vegetable,
- exemplar-based theory: specific instances, or *exemplars*, act as the prototypes against which other members are compared. Objects are grouped, relative to one another, based on some similarity metric. The exemplar-based theory differs from the prototype theory in that specific instances are the norm against which membership is decided. When asked to name particular members of a category, the attributes of the exemplars are used as cues to retrieve other objects having similar attributes,
- explanation-based theory: there is an explanation for why categories have the members they do. For instance, the biblical classification of food into *clean* and *unclean* is roughly explained by the correlation between type of habitat, biological structure, and form of locomotion; creatures of the sea should have fins, scales and swim (sharks and eels don't), and creatures of the land should have four legs (ostriches don't).

  From a predictive point of view, explanation-based categories suffer from the problem that they may heavily depend on the knowledge and beliefs of the person who formed the category; for instance, the set of objects a person would remove from their home, if it suddenly caught fire.

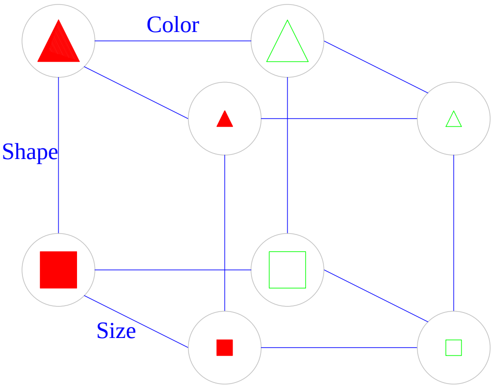


Figure 2.45: Orthogonal representation of shape, color and size stimuli. Based on Shepard.[1669]

Figure 2.45 shows the eight possible combinations of three, two-valued attributes, color/size/shape. It is possible to create six unique categories by selecting four items from these eight possibilities (see figure 2.46; there are 70 different ways of taking four things from a choice of eight, 8!/(4!4!), and taking symmetry into account reduces the number to unique categories to six).

A study by Shepard, Hovland, and Jenkins[1669] measured subject performance in assigning objects to these six categories. Subject error rate decreased with practice.

Estes[553] proposed the following method for calculating the similarity of two objects. Matching attributes have the same similarity coefficient, $0 \leq t \leq \infty$, and nonmatching attributes have similarity coefficient, $0 \leq s_i \leq 1$ (which is potentially different for each nonmatch). When comparing objects within the same category, the convention is to use $t = 1$, and to give the attributes that differ the same similarity coefficient, s.

Stimulus	Similarity to A	Similarity to B
Red triangle	$1+s$	$s+s^2$
Red square	$1+s$	$s+s^2$
Green triangle	$s+s^2$	$1+s$
Green square	$s+s^2$	$1+s$

Table 2.3: Similarity of a stimulus object to: category A: red triangle and red square; category B: green triangle and green square.

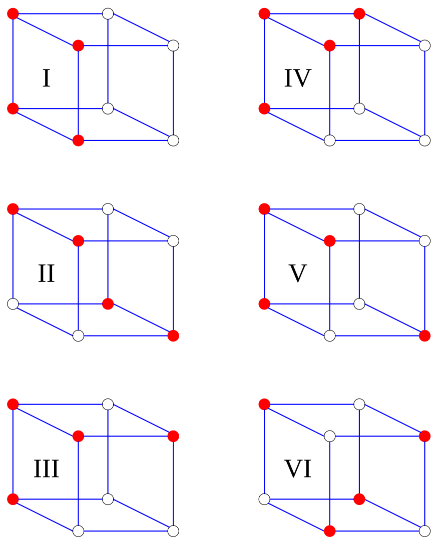


Figure 2.46: The six unique configurations of selecting four times from eight possibilities, i.e., it is not possible to rotate one configuration into another within these six configurations. Based on Shepard.[1669]

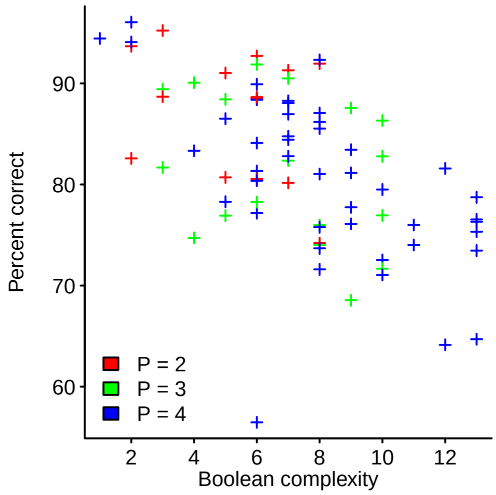


Figure 2.47: Percentage of correct category answers produced by one subject against boolean-complexity, broken down by number of positive cases needed to define the category used in the question (three colors). Data kindly provided by Feldman.[580] Github–Local

As an example, consider just two attributes shape/color in figure 2.45, giving the four combinations red/green—triangles/squares; assign red-triangle and red-square to category A, assign green-triangle and green-square to category B, i.e., category membership is decided by color. Table 2.3 lists the similarity of each of the four possible object combinations to category A and B. Looking at the top row: red-triangle is compared for similarity to all members of category A ($1+s$, because it does not differ from itself and differs in one attribute from the other member of category A), and all members of category B ($s+s^2$, because it differs in one attribute from one member of category B and in two attributes from the other member).

If a subject is shown a stimulus that belongs in category A, the expected probability of them assigning it to this category is: $\frac{1+s}{(1+s)+(s+s^2)} \rightarrow \frac{1}{1+s}$. When s is 1, the expected probability is no better than a random choice; when s is 0, the probability is a certainty.

A study by Feldman[580] investigated categories containing objects having either three or four attributes. During an initial training period subjects learned to distinguish between a creature having a given set of attributes, and other creatures that did not. Subjects then saw a sequence of example creatures, and had to decide whether they were a member of the learned category.

A later study[581] specified category membership algebraically, e.g., membership of category IV in the top right of figure 2.46 is specified by the expression: $S\overline{H}\overline{C}+SH\overline{C}+\overline{S}H\overline{C}+\overline{S}\overline{H}\overline{C}$, where: S is size, H is shape, C is color, and an $\overline{\textit{overline}}$ indicates negation. The number of terms in the minimal boolean formula specifying the category (a measure of category complexity, which Feldman terms *boolean complexity*) was found to predict the trend in subject error rate. Figure 2.47 shows, for one subject, the percentage of correct answers against boolean complexity; the number of positive cases needed to completely define the category of the animals in the question is broken down by color.

There are a few human generated semantic classification datasets publically available.[442]

2.5.4 Categorization consistency

Obtaining benefits from using categories requires some degree of consistency in assigning items to categories. In the case of an individual's internal categories, the ability to consistently assign items to the appropriate category is required; within cultural groups there has to be some level of agreement between members over the characteristics of items in each category.

Cross-language research has found that there are very few concepts that might be claimed to be universal (they mostly relate to the human condition).[1889, 1935]

Culture, and the use of items, can play an important role in the creation and use of categories.

A study by Bailenson, Shum, Atran, Medin and Coley[111] compared the categories created for two sets (US and Maya) of 104 bird species, by three groups; subjects were US bird experts (average of 22.4 years bird watching), US undergraduates, and ordinary Itzaj (Maya Amerindians people from Guatemala). The categorization choices made by the three groups of subjects were found to be internally consistent within each group. The US experts' categories correlated highly with the scientific taxonomy for both sets of birds, the Itzaj categories only correlated highly for Maya birds (they used ecological justifications; the bird's relationship with its environment), and the nonexperts had a low correlation for both set of birds. Cultural differences were found in that, for instance, US subjects were more likely to generalise from songbirds, while the Itzaj were more likely to generalize from perceptually striking birds.

A study by Labov[1053] showed subjects pictures of items that could be classified as either cups or bowls (see upper plot in figure 2.48; colors not in the original). These items were presented in one of two contexts—a neutral context in which the pictures were simply presented, and a food context (they were asked to think of the items as being filled with mashed potatoes).

The lower plot of figure 2.48 shows that as the width of the item seen was increased, an increasing number of subjects classified it as a bowl (dash lines). Introducing a food context, shifted subjects' responses towards classifying the item as a bowl at narrower widths (solid lines).

The same set of items may be viewed from a variety of different points of view (the term *frame* is sometimes used); for instance, commercial events include: buying, selling, paying, charging, pricing, costing, spending, and so on. Figure 2.49 shows four ways (i.e., buying, selling, paying, and charging) of classifying the same commercial event.

A study by Jones[923] investigated the extent to which different developers make similar decisions when creating data structures to represent the same information; see fig 12.5.

2.6 Reasoning

Reasoning enhances cognition. The knowledge-based use-case for the benefits of being able to reason, is the ability it provides to extract information from available data; adding constraints on the data (e.g., known behaviors) can increase the quantity of information that may be extracted. Dual process theories[1702, 1738, 1739] treat people as having two systems: unconscious reasoning and conscious reasoning. What survival advantages does an ability to consciously reason provide, or is it primarily an activity WEIRD people need to learn to pass school tests?

Outside of classroom problems, a real world context in which people explicitly reason is decision-making, and here "fast and frugal algorithms"[669, 672] provide answers quickly, within the limited constraints of time and energy. Context and semantics are crucial inputs to the reasoning process.[1751]

Understanding spoken language requires reasoning about what is being discussed,[1248] and in a friendly shared environment it is possible to fill in the gaps by assuming that what is said is relevant,[1725] with no intent to trick. In an adversarial context sceptical reasoning, of the mathematical logic kind, is useful for enumerating possible interpretations of what has been said.[1249]

The kinds of questions asked in studies of reasoning appear to be uncontentious. However, studies[1161, 1643] of reasoning using illiterate subjects, from remote parts of the world, received answers to verbal reasoning problems that were based on personal experience and social norms, rather than the western ideal of logic. The answers given by subjects, in the same location, who had received several years of schooling were much more likely to match those demanded by mathematical logic; the subjects had learned how to act WEIRD and *play the game*. The difficulties experienced by those learning formal logic suggests that there is no innate capacity for this task (innate capacity enables the corresponding skill to be learned quickly and easily). The human mind is a story processor, not a logic processor.[756]

Reasoning and decision-making appear to be closely related. However, reasoning researchers tied themselves to using the norm of mathematical logic for many decades, and created something of research ghetto,[1753] while decision-making researchers have been involved in explaining real-world problems.

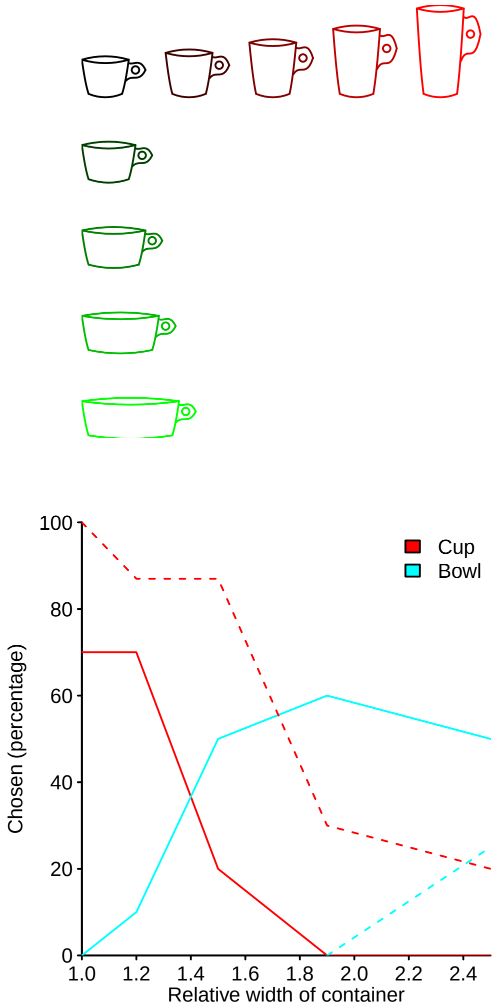


Figure 2.48: Cup- and bowl-like objects of various widths (ratios 1.2, 1.5, 1.9, and 2.5), and heights (ratios 1.2, 1.5, 1.9, and 2.4), with dashed lines showing neutral context and solid lines food context. The percentage of subjects who selected the term *cup* or *bowl* to describe the object they were shown (the paper did not explain why the figures do not sum to 100%, and color was not used in the original). Based on Labov.[1053] Github–Local

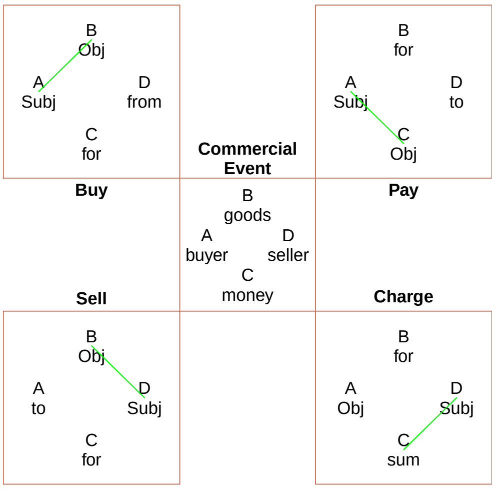


Figure 2.49: A commercial event involving a buyer, seller, money and goods; as seen from the buy, sell, pay, or charge perspective. Based on Fillmore.[593] Github–Local

The Wason selection task[1911] is to studies of reasoning like the fruit fly is to studies of genetics. Wason's study was first published in 1968, and considered mathematical logic to be the norm against which human reasoning performance should be judged. The reader might like to try this selection task:

- Figure 2.50 depicts a set of four cards, of which you can see only the exposed face but not the hidden back. On each card, there is a number on one side and a letter on the other.
- Below there is a rule, which applies only to these four cards. Your task is to decide, which, if any, of these four cards you must turn in order to decide whether the rule is true.
- Don't turn unnecessary cards. Specify the cards you would turn over.

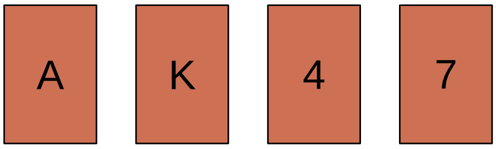


Figure 2.50: The four cards used in the Wason selection task. Based on Wason.[1911] Github–Local

Rule: If there is a vowel on one side, then there is an even number on the other side.

Answer: ?

The failure of many subjects to give the expected answer [ix] (i.e., the one derived using mathematical logic) surprised many researchers, and over the years a wide variety of explanations, experiments, thesis, and books have attempted to explain the answers given. Explanations for subject behavior include: human reasoning is tuned for detecting people who cheat[399] within a group where mutual altruism is the norm, interpreting the wording of questions pragmatically based on how natural language is used rather than as logical formula[822] (i.e., assuming a social context; people are pragmatic virtuosos rather than logical defectives), and adjusting norms to take into account cognitive resource limitations (i.e., computational and working memory), or a rational analysis approach.[1379]

Some Wason task related studies used a dialogue protocol (i.e., subjects' discuss their thoughts about the problem with the experimenter), and transcriptions[1752] of these studies read like people new to programming having trouble understanding what it is that they have to do to solve a problem by writing code.

> Alfred North Whitehead: "It is a profoundly erroneous truism . . . that we should cultivate the habit of thinking of what we are doing. The precise opposite is the case. Civilization advances by extending the number of important operations which we can perform without thinking about them."

People have different aptitudes, and this can result in them using different strategies to solve the same problem,[1738] e.g., an interaction between a subject's verbal and spatial ability, and the strategy used to solve linear reasoning problems.[1755] However, a person having high spatial ability, for instance, does not necessarily use a spatial strategy. A study by Roberts, Gilmore and Wood[1569] asked subjects to solve what appeared to be a spatial problem (requiring the use of a very inefficient spatial strategy to solve). Subjects with high spatial ability used non-spatial strategies, while those with low spatial ability used a spatial strategy. The conclusion drawn was that those with high spatial ability were able to see the inefficiency of the spatial strategy, and selected an alternative strategy, while those with less spatial ability were unable to make this evaluation.

A study by Bell and Johnson-Laird[164] investigated the effect of kind of questions asked on reasoning performance. Subjects had to give a yes/no response to two kinds of questions, asking about what is possible or what is necessary. The hypothesis was that subjects would find it easier to infer a *yes* answer to a question about what is possible, compared to one about what is necessary; because only one instance needs to be found for the possible question, whereas all instances need to be checked, to answer *yes* to a question about necessity. For instance, in a game in which only two can play, and the following information:

```
If Allan is in then Betsy is in.
If Carla is in then David is out.
```

answering *yes* to the question: "Can Betsy be in the game?" (a possibility), is easier than giving the same answer: to "Must Betsy be in the game?" (a necessity); see table 2.4.

Question	**Correct *yes***	**Correct *no***
is possible	91%	65%
is necessary	71%	81%

Table 2.4: Percentage of correct answers to possible/necessary questions, and the two kinds of response. Data from Bell et al.[164]

However, subjects would be expected to find it easier to infer a *no* answer to a question about what is necessary, compared to one about what is possible; because only one instance needs to be found to answer a necessary question, whereas all instances need to be

[ix] The letter A is a confirmation test, while the number 7 is a disconfirmation test.

checked to answer *no* to a question about possibility. For instance, in another two-person game, and the following information:

```
If Allan is out then Betsy is out.
If Carla is out then David is in.
```

answering *no* to the question: “Must Betsy be in the game?” (a necessity), is easier than giving the same answer to: “Can Betsy be in the game?” (a possibility); see table 2.4.

Conditionals in English: in human languages the conditional clause generally precedes the conclusion, in a conditional statement.[727] An example where the conditional follows the conclusion is: “I will leave, if you pay me”, given as the answer to the question: “Under what circumstances will you leave?”. In one study of English,[616] the conditional preceded the conclusion in 77% of written material, and 82% of spoken material. There is a lot of variation in the form of the conditional.[189, 297]

2.6.1 Deductive reasoning

Deductive reasoning is the process of reasoning about one or more statements (technically known as *premises*) to reach one or more logical conclusions.

Studies[106] have found that the reasoning strategies used by people involve building either a verbal or spatial model, from the premises. Factors found to affect peoples performance in solving deductive reasoning problems include the following:

- Belief bias: people are more willing to accept a conclusion, derived from given premises, that they believe to be true than one they believe to be false. A study by Evans, Barston, and Pollard[556] gave subjects two premises and a conclusion, and asked them to state whether the conclusion was true or false (based on the premises given; the conclusions were rated as either believable or unbelievable by a separate group of subjects); see the results in table 2.5,

Status-context	**Example**	**Accepted**
Valid-believable		
	No Police dogs are vicious Some highly trained dogs are vicious Therefore, some highly trained dogs are not police dogs	88%
Valid-unbelievable		
	No nutritional things are inexpensive Some vitamin tablets are inexpensive Therefore, some vitamin tablets are not nutritional things	56%
Invalid-believable		
	No addictive things are inexpensive Some cigarettes are inexpensive Therefore, some addictive things are not cigarettes	72%
Invalid-unbelievable		
	No millionaires are hard workers Some rich people are hard workers Therefore, some millionaires are not rich people	13%

Table 2.5: Percentage of subjects accepting that the stated conclusion could be logically deduced from the given premises. Based on Evans et al.[556]

Studies have modeled the response behavior using multinomial processing trees[1006] and system detection theory;[1824] see Github–developers/Trippas-2018.R.

- Form of premise: a study by Dickstein[488] measured subject performance on the 64 possible two premise syllogisms (a premise being one of the propositions: *All S are P*, *No S are P*, *Some S are P*, and *Some S are not P*). For instance, the following syllogisms show the four possible permutations of three terms (the use of S and P is interchangeable):

```
All M are S    All S are M    All M are S    All S are M
No P are M     No P are M     No M are P     No M are P
```

The results found that performance was affected by the order in which the terms occurred in the two premises of the syllogism. The order in which the premises are processed may affect the amount of working memory needed to reason about the syllogism, which in turn can affect human performance.[675]

2.6.2 Linear reasoning

Being able to make relational decisions is a useful skill for animals living in hierarchical social groups, where aggression is sometimes used to decide status.[1437] Aggression is best avoided, as it can lead to death or injury; the ability to make use of relative dominance information (obtained by watching interactions between other members of the group) may remove the need for aggressive behavior during an encounter between two group members who have not recently contested dominance (i.e., there is nothing to be gained in aggression towards a group member who has recently been seen to dominate a member who is dominant to yourself).

Another benefit of being ability to make relational comparisons is being able to select which of two areas contains the largest amount of food. Some animals, including humans, have a biologically determined representation of numbers, including elementary arithmetic operations, what one researcher has called the *number sense*.[467]

The use of relational operators have an interpretation in terms of linear syllogisms. A study by De Soto, London, and Handel[450] investigated a task they called *social reasoning*, using the relations *better* and *worse*. Subjects were shown two relationship statements involving three people, and a possible conclusion (e.g., "Is Mantle worse than Moskowitz?"), and were given 10 seconds to answer "yes", "no", or "don't know". The British National Corpus[1094] lists *better* as appearing 143 times per million words, while *worse* appears under 10 times per million words, and is not listed in the top 124,000 most used words.

	Relationships	**Correct %**		**Relationships**	**Correct %**
1.	A is better than B		5.	A is better than B	
	B is better than C	60.5		C is worse than B	61.8
2.	B is better than C		6.	C is worse than B	
	A is better than B	52.8		A is better than B	57.0
3.	B is worse than A		7.	B is worse than A	
	C is worse than B	50.0		B is better than C	41.5
4.	C is worse than B		8.	B is better than C	
	B is worse than A	42.5		B is worse than A	38.3

Table 2.6: Eight sets of premises describing the same relative ordering between A, B, and C (peoples names were used in the study) in different ways, followed by the percentage of subjects giving the correct answer. Based on De Soto et al.[450]

Table 2.6 shows the percentage of correct answers: a higher percentage of correct answers were given when the direction was better-to-worse (case 1), than mixed direction (cases 2 and 3); the consistent direction worse-to-better performed poorly (case 4); a higher percentage of correct answers were given when the premises stated an end term (better or worse; cases 1 and 5) followed by the middle term, than a middle term followed by an end term.

A second experiment, in the same study, gave subjects printed statements about people. For instance, "Tom is better than Bill". The relations used were *better*, *worse*, *has lighter hair*, and *has darker hair*. The subjects had to write the peoples names in two of four possible boxes; two arranged horizontally and two arranged vertically.

The results found 84% of subjects selecting a vertical direction for better/worse, with better at the top (which is consistent with the *up is good* usage found in English metaphors[1058]). In the case of lighter/darker, 66% of subjects used a horizontal direction, with no significant preference for left-to-right or right-to left.

A third experiment in the same study used the relations *to-the-left* and *to-the-right*, and *above* and *below*. The above/below results were very similar to those for better/worse. The left-right results found that subjects learned a left-to-right ordering better than a right-to-left ordering.

Subject performance on linear reasoning improves, the greater the distance between the items being compared; the *distance effect* is discussed in section 2.7.

Source code constructs relating to linear reasoning are discussed in section 7.1.2.

2.6.3 Causal reasoning

A question asked by developers, while reading source, is “what causes this /situation/event to occur?” Causal questions, such as this, are also occur in everyday life. However, there has been relatively little mathematical research on causality (statistics deals with correlation; Pearl[1439] covers some mathematical aspects of causality), and little psychological research on causal reasoning.[1705]

It is sometimes possible to express a problem in either a causal or conditional form. A study by Sloman, and Lagnado[1706] gave subjects one of the following two reasoning problems, and associated questions:

- Causal argument form:

  ```
  A causes B
  A causes C
  B causes D
  C causes D
  D definitely occurred
  ```

  with the questions: “If B had not occurred, would D still have occurred?”, or “If B had not occurred, would A have occurred?”

- Conditional argument form:

  ```
  If A then B
  If A then C
  If B then D
  If C then D
  D is true
  ```

  with the questions: “If B were false, would D still be true?”, or “If B were false, would A be true?”.

Table 2.7 shows that subject performance depended on the form in which the problem was expressed.

Question	Causal	Conditional
D holds?	80%	57%
A holds?	79%	36%

Table 2.7: Percentage “yes” responses to various forms of questions (based on 238 responses). Based on Sloman et al.[1706]

A study by Bramley[237] investigated causal learning. Subjects saw three nodes (e.g., grey filled circles, such as those at the center of figure 2.51), and were told that one or more causal links existed between the nodes. Clicking on a node activated it, and clicking the test icon resulted in zero or more of the other two nodes being activated (depending on the causal relationship that existing between nodes; nodes had to be activated to propagate an activation); subjects were told that the (unknown to them) causal links worked 80% of the time, and in 1% of cases a node would independently activate. Subjects, on Mechanical Turk, were asked to deduce the causal links that existed between the three nodes, by performing 12 tests for each of the 15 problems presented.

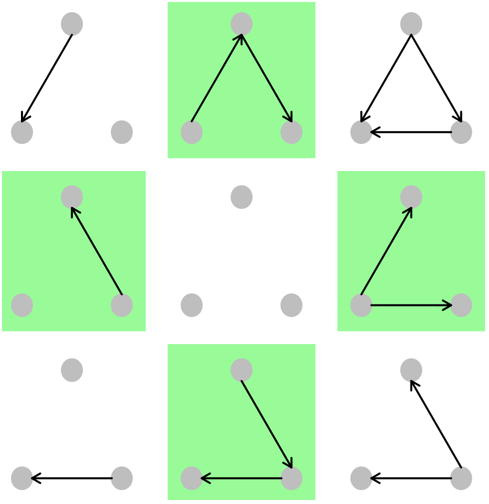


Figure 2.51: Example causal chains used Bramley.[237] Github–Local

Figure 2.51 shows some possible causal links, e.g., for the three nodes in the top left, clicking the test icon when the top node was activated would result in the left/below node becoming activated (80% of the time).

On average, subjects correctly identified 9 (sd=4.1) out of 15 causal links, with 34% getting 15 out of 15. A second experiment included on-screen reminder information and a summary of previous test results; the average score increased to 11.1 (sd=3.5), and 12.1 (sd=2.9) when previous test results were on-screen.

Common mistakes included: a chain $[X_1 \rightarrow X_2 \rightarrow X_3]$ being mistakenly judged to be fully connected $[X_1 \rightarrow X_2 \rightarrow X_3, X_1 \rightarrow X_3]$, or a fork $[X_2 \leftarrow X_1 \rightarrow X_3]$; the opposite also occurred, with fully connected judged to be a chain.

2.7 Number processing

Having a sense of quantity, and being able to judge the relative size of two quantities, provides a number of survival benefits, including deciding which of two clusters of food is the largest, and being able to repeat a task an approximate number times (e.g., pressing a bar is a common laboratory task, see fig 2.4).

Being able to quickly enumerate small quantities is sufficiently useful for the brain to support preattentive processing of up to four, or so, items.[1822] When asked to enumerate how many dots are visible in a well-defined area, subjects' response time depends on the number of dots; with between one and four dots, performance varies between 40 ms to 100 ms per dot, but with five or more dots, performance varies between 250 ms to 350 ms per dot. The faster process is known as *subitizing* (people effortlessly **see** the number of dots), while the slower process is called *counting*.

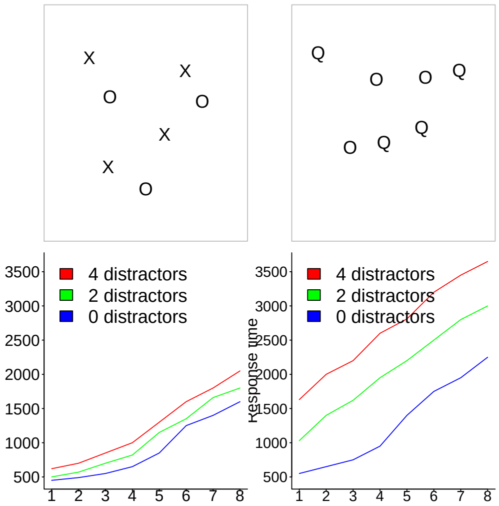


Figure 2.52: Average time (in milliseconds) taken for subjects to enumerate O's in a background of X or Q distractors. Based on Trick and Pylyshyn.[1822] Github–Local

Studies have found that a variety of animals make use of an approximate mental number system (sometimes known as the *number line*); see fig 2.4. The extent to which brains have a built-in number line, or existing neurons are repurposed through learning, is an active area of research.[465, 1376] Humans are the only creatures known to have a second system, one that can be used to represent numbers exactly: language.

A study by van Oeffelen and Vos[1859] investigated subjects' ability to estimate the number of dots in a briefly displayed image (100 ms, i.e., not enough time to be able to count the dots). Subjects were given a target number, and had to answer yes/no on whether they thought the image they saw contained the target number of dots. Figure 2.53 shows the probability of a correct answer for various target numbers, and a given difference between target number and number of dots displayed.

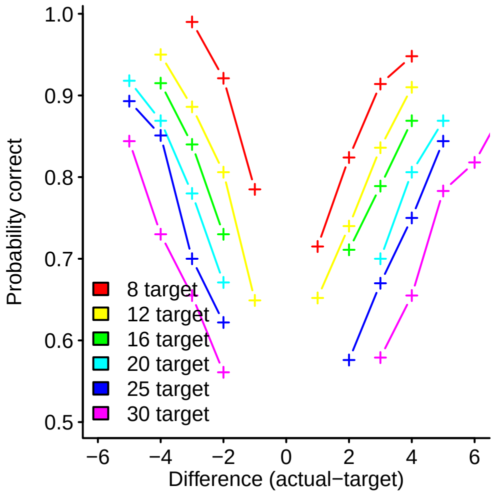


Figure 2.53: Probability a subject will successfully distinguish a difference between the number of dots displayed, and a specified target number (x-axis is the difference between these two values). Data extracted from van Oeffelen et al.[1859] Github–Local

What are the operating characteristics of the approximate number system? The characteristics that have most occupied researchers are the scale used (e.g., linear or logarithmic), the impact of number magnitude on cognitive performance, and when dealing with two numbers the effect of their relative difference in value on cognitive performance.[467]

Studies of the mental representation of single digit numbers[470] have found a linear scale used by subjects from western societies, and a logarithmic scale used by subjects from indigenous cultures that have not had formal schooling.

Engineering and science sometimes deal with values spanning many orders of magnitude, a range that people are unlikely to encounter in everyday life. How do people mentally represent large value ranges?

A theoretical analysis[1465] found that a logarithmic scale minimized representation error, assuming the probability of a value occurring follows a power law distribution, assuming relative change is the psychologically-relevant measure, and that noise is present in the signal.

A study by Landy, Charlesworth and Ottmar[1067] asked subjects to click the position on a line (labeled at the left end with one thousand and at the right end with one billion), corresponding to what they thought was the appropriate location for each of the 182 values they saw (selected from 20 evenly spaced values between one thousand and one million, and 23 evenly spaced values between one million and one billion).

Figure 2.54 shows some of the patterns that occurred in subject responses; the one in the top left was one of the most common. Most subjects placed one million at the halfway point (as-if using a logarithmic scale), placing values below/above a million on separate linear scales. Landy et al developed a model based on the Category Adjustment model,[505] where subjects selected a category boundary (e.g., one million, creating the categories: the thousands and the millions), a location for the category boundary along the line, and a linear(ish)[x] mapping of values to relative position within their respective category.

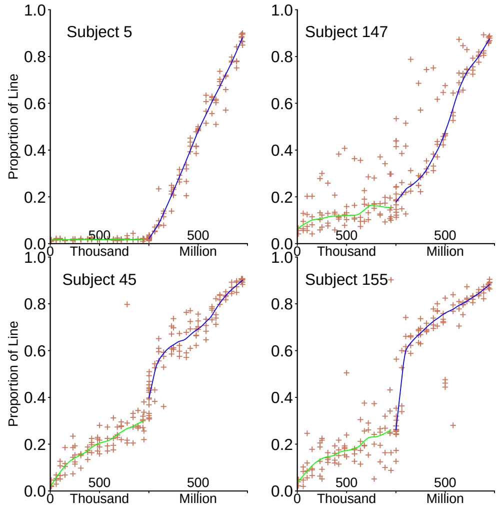


Figure 2.54: Line locations chosen for the numeric values seen by each of four subjects; color of fitted loess line changes at one million boundary. Data kindly provided by Landy.[1067] Github–Local

Studying the learning and performance of simple arithmetic operations has proven complicated;[608, 1862] models of simple arithmetic performance[1088] have been built. Working memory capacity has an impact on the time taken to perform mental arithmetic operations, and the likely error rate, e.g., remembering carry or borrow quantities during subtraction;[878] the human language used to think about the problem also has an impact.[877]

How do people compare multi-digit integer constants? For instance, do they compare them digit by digit (i.e., a serial comparison), or do they form two complete values before comparing their magnitudes (the so-called *holistic* model)? Studies show that the

[x] Category Adjustment theory supports curvaceous lines.

answer depends on how the comparisons are made, with results consistent with the digit by digit[1987] and holistic[469] approaches being found.

Other performance related behaviors include:

- *split effect*: taking longer to reject false answers that are close to the correct answer (e.g., $4 \times 7 = 29$), than those that are further away (e.g., $4 \times 7 = 33$),
- *associative confusion* effect: answering a different question from the one asked (e.g., giving 12 as the answer to $4 \times 8 =?$, which would be true had the operation been addition),
- plausibility judgments:[1101] using a rule, rather than retrieving a fact from memory, to verify the answer to a question; for instance, adding an odd number to an even number always produces an odd result,

Trial judges have been found[1529] to be influenced by the unit of measurement in sentencing (i.e., months or years), and to exhibit anchoring effects when deciding award damages.

2.7.1 Numeric preferences

Measurements of number usage, in general spoken and written form, show that people prefer to use certain values, either singly (sometimes known as *round numbers*) or as number pairs.

Number pairs (e.g., "10 to 15 hours ago") have been found to follow a small set of rules,[547] including: the second number is larger than the first, the difference between the values is a divisor of the second value, and the difference is at least 5% of the second value.

A round number is any number commonly used to communicate an approximation of nearby values; round numbers are often powers of ten, divisible by two or five, and other pragmatic factors.[903] Round numbers can act as goals[1485] and as clustering points.[1716]

A usage analysis[141] shows that selecting a rounded interpretation yields the greater benefit when there is a small chance that a rounded, rather than or non-rounded, use occurred. If a speaker uses a round number, *round_value*, the probability that the speaker rounded a nearby value to obtain it is given by:

$P(Speaker\_rounded|round\_value) = \dfrac{k}{k+\frac{1}{x}-1}$, where: k is the number of values likely to be rounded to *round_value*, and x the probability that the speaker chooses to round. Figure 2.55 shows how the likelihood of rounding being used increases rapidly as the number of possible rounded values increases.

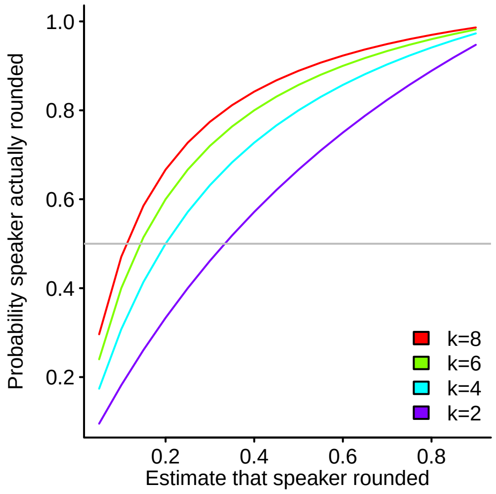


Figure 2.55: Probability the rounded value given has actually been rounded, given an estimate of the likelihood of rounding, and the number of values likely to have been rounded; grey line shows 50% probability of rounding. Github–Local

A study by Basili, Panlilio-Yap, Ramsey, Shih and Katz[137] investigated the redesign and implementation of a software system. Figure 2.56 shows the number of change requests taking a given amount of time to decide whether the change was needed, and the time to design+implement the change. There are peaks at the round-numbers 1, 2 and 5, with 4 hours perhaps being the Parkinson's law target of a half day.

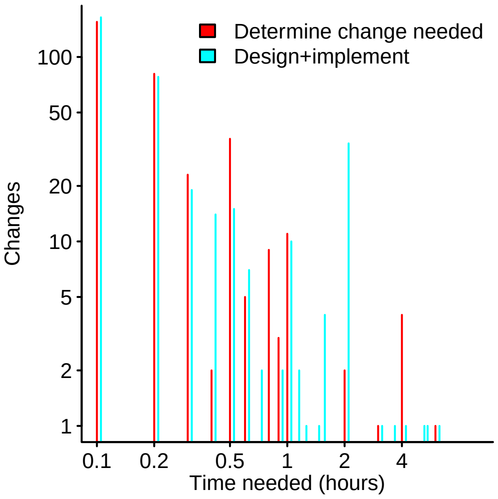


Figure 2.56: Number of change requests having a given recorded time to decide whether change was needed, and time to implement. Data from Basili et al.[137] Github–Local

A study by King and Janiszewski[991] showed integer values between 1 and 100, in random order, to 360 subjects, asking them to specify whether they liked the number, disliked it, or were neutral. Numbers ending in zero had a much higher chance of being liked, compared to numbers ending in other digits; see Github–developers/like-n-dis.R for a regression model fitted to the like and dislike probability, based on the first/last digits of the number.

A study by van der Henst, Carles and Sperber[1854] approached people on the street, and asked them for the time; there was a 20% probability that the minutes were a multiple of five. When subjects were wearing an analogue watch, 98% of replies were multiples of five-minutes, when subjects were wearing digital watches the figure was 66%. Many subjects invested effort in order to communicate using a round number.

People sometimes denote approximate values using numerical expressions containing comparative and superlative qualifiers, such as "more than n" and "at least n".

A study by Cummins[414] investigated the impact of number granularity on the range of values assigned by subjects to various kinds of numerical expressions. Subjects saw statements of the form "So far, we've sold fewer than 60 tickets." (in other statements fewer was replaced by more), and subjects were asked: "How many tickets have been sold? From ?? to ??, most likely ??.". Three numeric granularities were used: *coarse*,

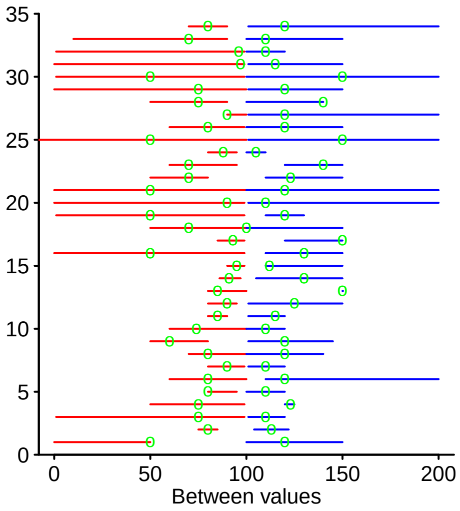


Figure 2.57: Min/max range of values (red/blue lines), and best value estimate (green circles), given by subjects interpreting the value likely expressed by statements containing "less than 100" and "more than 100". Data kindly provided by Cummins.[414] Github–Local

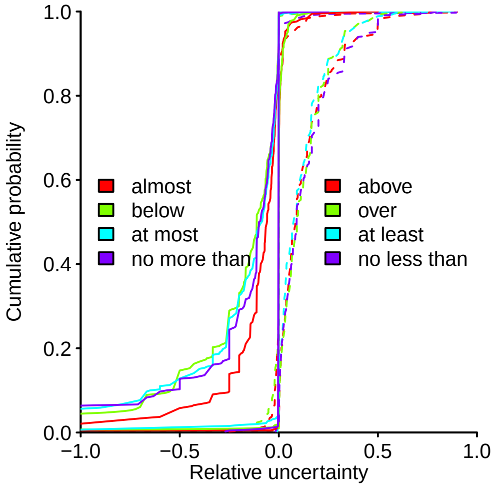


Figure 2.58: The cumulative probability of subjects expressing a given relative uncertainty, for numeric phrases using given hedge words. Data kindly provided by Ferson.[590] Github–Local

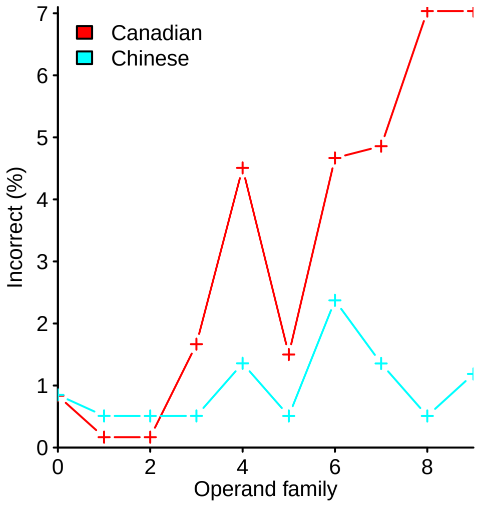


Figure 2.59: Percentage of incorrect answers to arithmetic problems, given by Canadian and Chinese students, for each operand family value. Data kindly provided by LeFevre.[1096] Github–Local

e.g., a multiple of 100, *medium* e.g., multiple of 10 and non-multiple of 100, and *fine* e.g., non-round such as 77.

The results found that the *most likely* value, given by subjects, was closest to the value appearing in the statement when it had a *fine* granularity and furthest away when it was *coarse*; see Github–reliability/CumminsModifiedNumeral.R. Figure 2.57 shows the "From to" range given by each subject, along with their best estimate (in green) for statements specifying "less than 100" and "more than 100".

When specifying a numeric value, people may also include information that expresses their uncertainty, using a so-called *hedge* word, e.g., "about", "around", "almost", "almost exactly", "at least", "below" and "nearly".

A study by Ferson, O'Rawe, Antonenko, Siegrist, Mickley, Luhmann, Sentz and Finkel[590] investigated interpretations of numerical uncertainty. Subjects were asked to specify minimal and maximal possible values, for their interpretation of the quantity expressed in various statements, such as: "No more than 100 people."

Figure 2.58 shows the cumulative probability of relative uncertainty of numeric phrases (calculated as: $\frac{minimal-actual}{minimal}$ and $\frac{maximal-actual}{maximal}$), for the hedge words listed in the legends.

The relative size of objects being quantified has been found to have an effect on the interpretation given to quantifiers.[1351]

2.7.2 Symbolic distance and problem size effect

When people compare two items sharing a distance-like characteristic, the larger the distance between two items, the faster people are likely to respond to a comparison question involving this characteristic (e.g., comparisons of social status[343] and geographical distance;[825] also see fig 2.19); this is known as the *symbolic distance effect*. Inconsistencies between a symbolic distance characteristic and actual distance for the question asked, can increase the error rate,[798] e.g., is the following relation true? 3 >5.

A study by Tzelgov, Yehene, Kotler and Alon[1839] investigated the impact of implicit learning on the symbolic distance effect. Subjects trained one-hour per day on six different days (over ten days) learning the relative order of pairs of nine graphical symbols. During the first three training sessions all subjects only saw pairs of symbols that were adjacent in the overall relative ordering, i.e., pairs: (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8) and (8, 9). During the next three sessions one group of subjects continued to train on the same pairs, while another group trained on 14 non-adjacent pairs (i.e., a subset of possible non-adjacent pairs).

On completion of the training sessions, subjects answered questions about pairs of symbols having various relative ordering distances. A symbolic distance effect was seen in answer response times; subjects who had trained on non-adjacent pairs consistently responded faster than subjects who had only trained on adjacent pairs.

Studies have found that the time taken to solve a simple arithmetic problem, and the error rate, increase as the value of both operands increases (e.g., subjects are slower to solve $9+7$, than $2+3$,[287] see Github–developers/campbell1997.R); this is known as the *problem size effect*. As in many experiments, the characteristics of student performance can vary widely.

A study by LeFevre and Liu[1096] investigated the performance of Canadian and Chinese university students on simple multiplication problems (e.g., 7×4). Figure 2.59 shows the percentage error rate for problems containing values from a given operand family (e.g., 2×6 contains values from the 2 and 6 operand family, and an incorrect answer to this problem counts towards both operand families). The hypothesis for the difference in error rate was student experience; during their education Canadian students were asked to solve more multiplication problems containing small values, compared to large values; an analysis of the work-books of Chinese students found the opposite distribution of operand values, i.e., Chinese students were asked to solve more problems involving large operands, compared to small operands.

The magnitude of a measurement value depends on the unit of measurement used, e.g., inches, feet, yards, and meters are units of length, or distance.

A study Jørgensen[937] asked commercial developers to estimate the time needed to implement a project. Some were asked to estimate in work-hours and others in work days.

The results found that those answering in work-hours gave lower estimates, for the same project, than those working in workdays; see Github–developers/time-unit-effect.R.

2.7.3 Estimating event likelihood

The ability to detect regularities in the environment, and to take advantage of them is a defining characteristic of intelligent behavior.

In some environments, the cost of failing to correctly detect an object or event may be significantly higher than the cost of acting as-if an event occurred, when none occurred. People appear to be hardwired to detect some patterns, e.g., seeing faces in a random jumble of items.

People sometimes have expectations of behavior for event sequences. For instance, an expectation that sequences of random numbers have certain attributes,[562, 1405] e.g., frequent alternation between different values (which from a mathematical perspective, are a form of regularity).

People tend to overestimate the likelihood of uncommon events and underestimate the likelihood of very common events. A variety of probability weighting functions have been proposed to explain the experimental evidence.[695] The log-odds of the proportion estimated, $\log \frac{p_e}{1-p_e} \Rightarrow lo_{pe}$ (where p_e is the proportion estimated), is given by:

$lo_{pe} = \gamma lo_{pa} + (1-\gamma)\delta$, where: γ is a constant (interpreted as a relative weight on the perception of lo_{pa}, the log-odds of the actual proportion), and δ is a constant (interpreted as a prior expectation).[xi]

Figure 2.60 shows data from various surveys of public perception of demographic questions involving the percentage of a population containing various kinds of people, along with a fitted log-odds proportional model (grey line shows estimated equal actual).

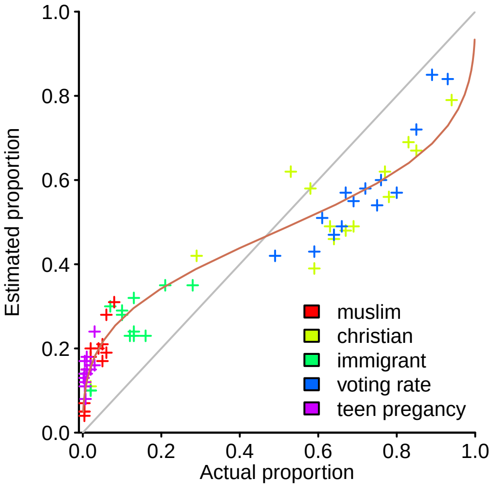


Figure 2.60: Estimated proportion (from survey results), and actual proportion of people in a population matching various demographics; line is a fitted regression having the form: $lo_{Estimated} \propto \gamma \times lo_{Actual} + (1-\gamma) \times \delta$, where γ and δ are fitted constants; grey line shows estimated equals actual. Data from Landy et al.[1069] Github–Local

The probability of a particular event occurring may not be fixed, the world is constantly changing, and the likelihood of an event may change. Studies have found that people are responsive to changes in the frequency with which an event occurs,[639] and to changes in the available information.[1093]

A study by Khaw, Stevens and Woodford[982] investigated changes in decision-maker answers when operating in a changing environment. Subjects were asked to estimate the likelihood of drawing a green ring from a box containing an unknown (to them) percentage of red and green rings; the percentage of color rings remained fixed for some number of draws, and then changed. During a session each subject made 999 draws and estimates of probability of drawing a green ring. Subjects were paid an amount based on how accurately they estimated, over time, the percentage of green rings, with eleven subjects each performing ten sessions.

The results found discrete jumps in subject estimates (rather than a continual updating of probability as each draw revealed more information, as a Bayesian decision-maker would behave), and a preference for round numbers.

Figure 2.61 shows two subjects' estimates (blue/green lines) of the probability of drawing a green ring, for a given actual probability (red line), as they drew successive rings.

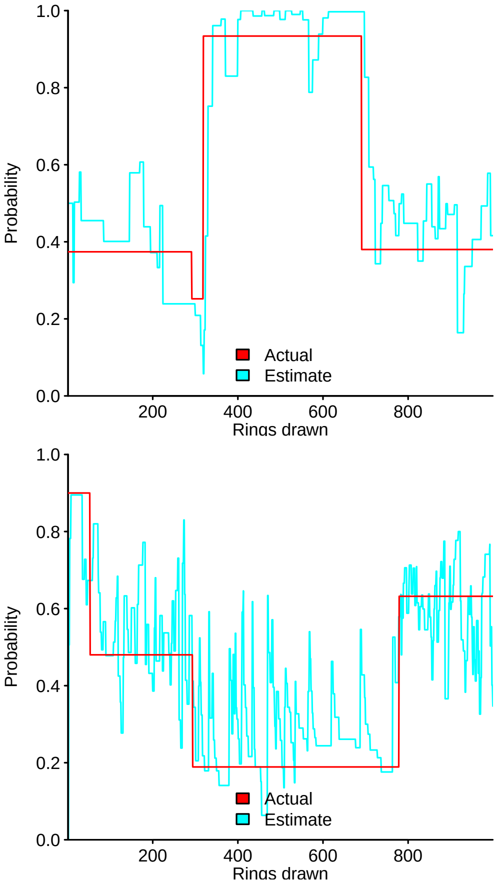


Figure 2.61: Estimated probability (blue/green lines) of drawing a green ring by two subjects (upper: subject 10, session 8, lower: subject 7, session 8), with actual probability in red. Data from Khaw et al.[982] Github–Local

A change in the occurrence of an event can result in a change to what is considered to be an occurrence of the event. A study by Levari, Gilbert, Wilson, Sievers, Amodio and Wheatley[1106] showed subjects a series of randomly colored dots, one at a time (the colors varied on a continuous scale, between purple and blue); subjects were asked to judge whether each dot they saw was blue (each subject saw 1,000 dots). Subjects were divided into two groups: for one group the percentage of blue dots did not change over time, while for the second group the percentage of blue dots was decreased after a subject had seen 200 dots.

Figure 2.62 is based on data from the first and last series of 200 dots seen by each subject; the x-axis is an objective measure of the color of each dot, the y-axis is the percentage of dots identified as blue (averaged over all subjects). Lines are fitted logistic regression models. The responses for subjects in the constant blue group did not change between the first/last 200 dots (red and green lines). For the decreased blue group, after subjects experienced a decline in the likelihood of encountering a blue dot, the color of what they considered to be a blue dot shifted towards purple (blue and purple lines).

[xi] This equation is sometimes written using p_a directly, rather than using log-odds, i.e., $p_e = \frac{\delta^{1-\gamma} p_a^{\gamma}}{\delta^{1-\gamma} p_a^{\gamma} + (1-p_a)^{\gamma}}$.

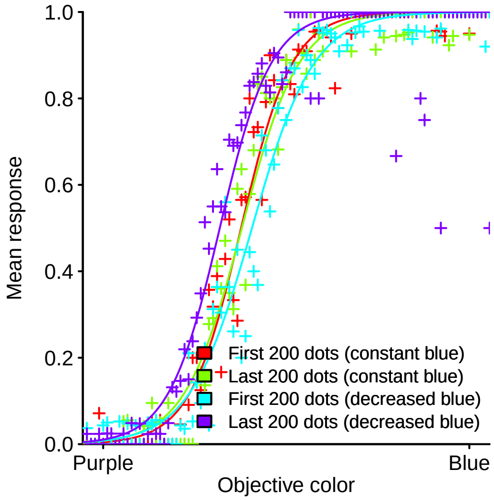


Figure 2.62: Mean likelihood that a subject considered a dot of a given color to be blue, for the first/last 200 dots seen by two groups of subjects; lines are fitted logistic regression models. Data from Levari et al.[1106] Github–Local

In later experiments subjects were told that the prevalence of blue dots "might change", and would "definitely decrease"; the results were unchanged.

2.8 High-level functionality

This section discusses high-level cognitive functionality that has an impact on software development, but for which there is insufficient material for a distinct section.

2.8.1 Personality & intelligence

Perhaps the most well-known personality test is the *Meyer-Briggs type indicator*, or MBTI (both registered trademarks of Consulting Psychologists Press). The reliability and validity of this test has been questioned,[1219, 1472] with commercial considerations, and a changing set of questions making research difficult. The International Personality Item Pool (IPIP)[689] is a public domain measure, which now has available samples containing hundreds of thousands of responses.[909]

The Five-Factor model[688] structures personality around five basic traits: neuroticism, extroversion, openness, agreeableness, and conscientiousness; it has its critics.[1654] Studies that have attempted to identify personality types, based on these five traits have suffered from small sample size (e.g., 1,000 people); studies based on IPIP data, using samples containing 100,000 to 500,000 responses, claim to have isolated four personality types.[661]

Tests measuring something called IQ have existed for over 100 years. The results are traditionally encapsulated in a single number, but tests claiming to measure the various components of intelligence are available.[455] The model is that people possess a general intelligence *g*, plus various domain specific intelligences, e.g., in tests involving maths the results would depend on *g* and maths specific intelligence, and in tests involving use of language the results would depend on *g* and language specific intelligence.

The cultural intelligence hypothesis is that humans have a specific set of social skills for exchanging knowledge in social groups; children and chimpanzees have been found to have similar cognitive skills for dealing with the physical world, but children have more sophisticated skills for dealing with the social world.[809]

Without reliable techniques for measuring personality traits, it is not possible to isolate characteristics likely to be beneficial or detrimental to software development. For instance, how important is the ability to concentrate for large amounts of time on particular activities?

2.8.2 Risk taking

Making a decision based on incomplete or uncertain information involves an element of risk. How do people integrate risk into their decision-making process?

The term *risk asymmetry* refers to the fact that people have been found to be *risk averse* when deciding between alternatives that have a positive outcome, but are *risk seeking* when deciding between alternatives that have a negative outcome.[xii]

While there is a widespread perception that women are more risk averse than men, existing studies are either not conclusive or show a small effect[592] (many suffer from small sample sizes, and dependencies on the features of the task subjects are given). For the case of financial risks. the evidence[329] that men are more willing to take financial risks than women is more clear-cut. The evidence from attempts to improve road safety is that “protecting motorists from the consequences of bad driving encourages bad driving”.[8]

A study by Jones[925] investigated the possibility that some subjects, in an experiment involving recalling information about previously seen assignment statements, were less willing to risk giving an answer when they had opportunity to specify that in real-life they would refer back to previously read code. Previous studies[920, 921] had found that a small percentage of subjects consistently gave much higher rates of "would refer back" answers. One explanation is that these subjects had a smaller short term memory capacity

[xii] While studies[804] based on subjects drawn from non-WEIRD societies sometimes produce different results, this book assumes developers are WEIRD.

than other subjects (STM capacity varies between people), another is that these subjects are much more risk averse than the other subjects.

The Domain-Specific Risk-Taking (DOSPERT) questionnaire[205, 1917] was used to measure subject's risk attitude. The results found no correlation between risk attitude (as measured by DOSPERT), and number of "would refer back" responses.

2.8.3 Decision-making

A distinction is commonly made between decisions made under risk, and decisions made under uncertainty.[1888] Decisions are made under risk when all the information about the probability of outcomes is available, and one of the available options has to be selected. Many studies investigating human decision-making provide subjects with all the information about the probability of the outcomes, and ask them to select an option, i.e., they involve decision under risk.

Decisions are made under uncertainty when the available information is incomplete and possibly inaccurate. Human decision-making often takes place in an environment of incomplete information and limited decision-making resources (e.g., working memory capacity and thinking time); people have been found to adopt various strategies to handle this situation,[1192] balancing the predicted cognitive effort required to use a particular decision-making strategy against the likely accuracy achieved by that strategy.

The term *bounded rationality*[1689] is used to describe an approach to problem-solving that makes limited use of cognitive resources. A growing number of studies[672] have found that the methods used by people to make decisions, and solve problems, are often close enough to optimal [xiii], given the resources available to them; even when dealing with financial matters.[1224] If people handle money matters in this fashion, their approach to software development decisions is unlikely to fare any better.

A so-called *irregular choice* occurs when, a person who chooses `B` from the set of items `{A, B, C}` does not choose `B` from the set `{A, B}`; irregular decision makers have been found[1802] to be more common among younger (18–25) subjects, and are less common with older (60–75) subjects.

Consumer research into understanding how shoppers decide among competing products uncovered a set of mechanisms that are applicable to decision-making in general, e.g., decision-making around the question of which soap will wash the whitest is no different from the question of whether an **if** statement, or a **switch** statement should be used. Before a decision can be made, a decision-making strategy has to be selected, and people have been found to use a number of different decision-making strategies.[1436]

Decision strategies differ in several ways, for instance, some make trade-offs among the attributes of the alternatives (making it possible for an alternative with several good attributes to be selected, instead of the alternative whose only worthwhile attribute is excellent); they also differ in the amount of information that needs to be obtained, and the amount of cognitive processing needed to make a decision. Named decision-making strategies include: the weighted additive rule, the equal weight heuristic, the frequency of good and bad features heuristic, the majority of confirming dimensions heuristic, the satisficing heuristic the lexicographic heuristic, the habitual heuristic and in recent years decision by sampling.[1758]

There is a hierarchy of responses for how people deal with time pressure:[1436] they work faster; if that fails, they may focus on a subset of the issues; if that fails, they may change strategies (e.g., from alternative based to attribute based).

Studies have found that having to justify a decision can affect the choice of decision-making strategy used.[1804] One strategy for handling accountability is to select the alternative that the perspective audience is thought most likely to select.[1803] People who have difficulty determining which alternative has the greatest utility tend to select the alternative that supports the best overall reasons (for choosing it).[1691]

Requiring developers to justify why they have not followed existing practice can be a double-edged sword. Developers can respond by deciding to blindly follow everybody else (a path of least resistance), or they can invest effort in evaluating alternatives (not

[xiii] Some researchers interpret bounded rationality as human decision-making producing results as-if people optimize under cognitive constraints, while others[668] don't require people to optimize, but to use algorithms that produce results that are good enough.

necessarily cost effective behavior, since the invested effort may not be warranted by the expected benefits). The extent to which some people will blindly obey authority was chillingly demonstrated in a number of studies by Milgram.[1261]

Making decisions can be difficult, and copying what others do can be a cost effective strategy. Who should be copied? Strategies include: copy the majority, copy successful individuals, copy kin, copy "friends", copy older individuals. Social learning is discussed in section 3.4.4.

A study by Morgan, Rendell, Ehn, Hoppitt and Laland[1294] investigated the impact of the opinions expressed by others on the answers given by subjects. One experiment asked subjects to decide whether one object was a rotated version of a second object (i.e., the task shown in figure 2.8), and to rate their confidence in their answer (on a scale of 0 to 6, with 6 being the most confident). After giving an answer, subjects saw the number of yes/no answers given by up to twelve other people (these answers may or may not have been correct); subjects were then asked to give a final answer. The 51 subjects each completed 24 trials, and after seeing other responses changed their answer, on average, in 2.8 trials.

Figure 2.63 shows the behavior of a fitted regression model for the probability that a subject (y-axis), who makes three switches in 24 trials, does switch their answer, when told that a given fraction of other responses were the opposite of the subject's initial guess (x-axis); lines show the percentage for subject's expressing a given confidence in their answer (colored lines).

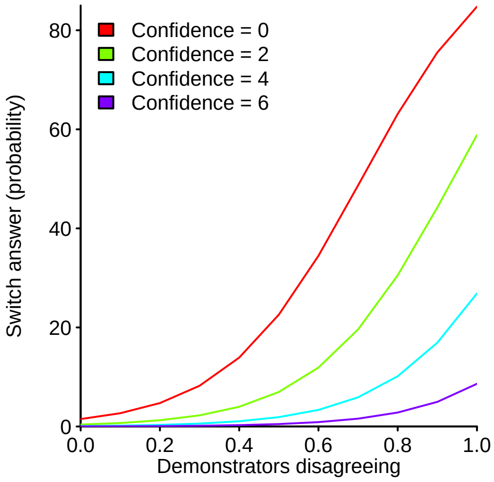


Figure 2.63: Fitted regression model for probability that a subject, who switched answer three times, switches their initial answer when told a given fraction of opposite responses were made by others (x-axis), broken down by confidence expressed in their answer (colored lines). Data kindly provided by Morgan.[1294] Github–Local

Social pressure can cause people to go along with decisions voiced by others involved in the decision process (i.e., social conformity as a signal of belonging,[194] such as corporate IT fashion: see fig 1.16). A study by Asch[78] asked groups of seven to nine subjects to individually state, which of three black strips they considered to be the longest (see figure 2.64). The group sat together in front of the stripes, and subjects could interact with each other; all the subjects, except one, were part of the experiment, and in 12 of 18 questions selected a stripe that was clearly not the longest (i.e., the majority gave an answer clearly at odds with visible reality). It was arranged that the actual subject did not have to give an answer until hearing the answers of most other subjects.

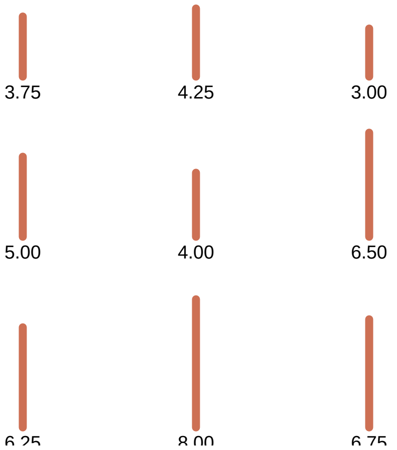


Figure 2.64: Each row shows a scaled version of the three stripes, along with actual lengths in inches, from which subjects were asked to select the longest. Based on Asch.[78] Github–Local

The actual subject, in 24% of groups, always gave the correct answer; in 27% of groups the subject agreed with the incorrect majority answer between eight and twelve times, and just under half varied in the extent to which they followed the majority decision. When the majority selected the most extreme incorrect answer (i.e., the shortest stripe), subjects giving an incorrect answer selected the less extreme incorrect answer in 20% of cases.

How a problem is posed can have a large impact on the decision made.

A study by Regnell, Höst, och Dag, Beremark and Hjelm[1548] asked 10 subjects to assign a relative priority to two lists of requirements (subjects' had a total budget of 100,000 units and had to assign units to requirements). One list specified that subjects should prioritize the 17 high level requirements, and the other list specified that a more detailed response, in the form of prioritizing every feature contained within each high level requirement, be given.

Comparing the totals for the 17 high level requirements list (summing the responses for the detailed list) with the more detailed response, showed that the subject correlation between the two lists was not very strong (the mean across 10 subjects was 0.46); see Github–projects/prioritization.R.

2.8.4 Expected utility and Prospect theory

The outcome of events is often uncertain. If events are known to occur with probability p_i, with each producing a value of X_i, then the expected outcome value is given by:

$$E[x] = p_1X_1 + p_2X_2 + p_3X_3 + \cdots + p_nX_n$$

For instance, given a 60% chance of winning £10 and a 40% chance of winning £20, the expected winnings are: $0.6 \times 10 + 0.4 \times 20 = 14$

When comparing the costs and benefits of an action, decision makers often take into account information on their current wealth, e.g., a bet offering a 50% chance of winning £1 million, and a 50% chance of losing £0.9 million has an expected utility of £0.05 million; would you take this bet, unless you were very wealthy? Do you consider £20 to be worth twice as much as £10?

The mapping of a decision's costs and benefits, to a decision maker's particular circumstances, is made by what is known as a *utility function*, u; the above equation becomes (where W is the decision maker's current wealth):

$$E[x] = p_1u(W+X_1) + p_2u(W+X_2) + p_3u(W+X_3) + \cdots + p_nu(W+X_n)$$

In some situations a decision maker's current wealth might be effectively zero, e.g., they have forgotten their wallet, or because they have no authority to spend company money on work related decisions (personal wealth of employees is unlikely to factor into many of their work decisions).

Figure 2.65 shows possible perceived utilities of an increase in wealth. A risk neutral decision maker perceives the utility of an increase in wealth as being proportional to the increase (green, $u(w) = w$), a risk loving decision maker perceives the utility of an increase in wealth as being proportionally greater than the actual increase (e.g., red, $u(w) = w^2$), while a risk averse decision maker perceives the utility of an increase having proportionally less utility (e.g., blue, $u(w) = \sqrt{w}$).

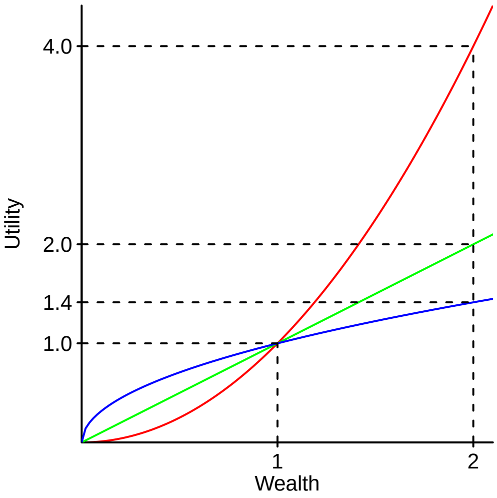


Figure 2.65: Risk neutral (green, $u(w) = w$), and example of risk loving (red, quadratic) and risk averse (blue, square-root) utility functions. Github–Local

A study by Kina, Tsunoda, Hata, Tamada and Igaki[990] investigated the decisions made by subjects (20 Open source developers) when given a choice between two tools, each having various probabilities of providing various benefits, e.g., $Tool_1$ which always saves 2 hours of work, or $Tool_2$ which saves 5 hours of work with 50% probability, and has no effect on work time with 50% probability.

Given a linear utility function, $Tool_1$ has an expected utility of 2 hours, while $Tool_2$ has an expected utility of 2.5 hours. The results showed 65% of developers choosing $Tool_1$, and 35% choosing $Tool_2$: those 65% were not using a linear utility function; use of a square-root utility function would produce the result seen, i.e., $1 \times \sqrt{2} > 0.5 \times \sqrt{5} + 0.5 \times \sqrt{0}$.

2.8.5 Overconfidence

While overconfidence can create unrealistic expectations, and lead to hazardous decisions being made (e.g., allocating insufficient resources to complete a job), simulation studies[908] have found that overconfidence has benefits in come situations, averaged over a population; see Github–developers/overconfidence/0909.R. A study[80] of commercialization of new inventions, found that while inventors are significantly more confident and optimistic than the general population, the likely return on their investment of time and money in their invention is negative; a few receive a huge pay-off.

A study by Lichtenstein and Fishhoff[1121] asked subjects general knowledge questions, with the questions divided into two groups, hard and easy. Figure 2.66 shows that subjects' overestimated their ability (x-axis) to correctly answer hard questions, but underestimated their ability to answer easy questions; green line denotes perfect self-knowledge.

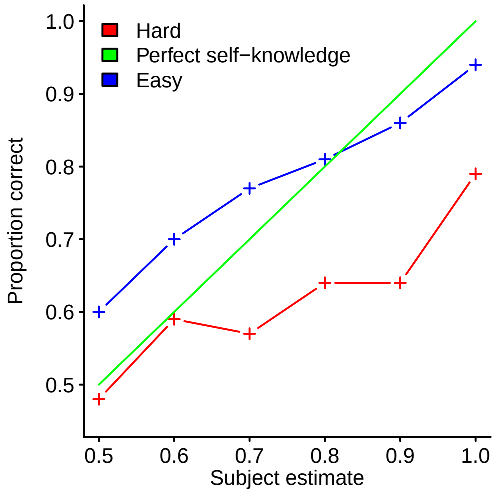


Figure 2.66: Subjects' estimate of their ability (x-axis) to correctly answer a question and actual performance in answering on the left scale. The responses of a person with perfect self-knowledge is given by the green line. Data extracted from Lichtenstein et al.[1121] Github–Local

These, and subsequent results, show that the skills and knowledge that constitute competence in a particular domain are the same skills needed to evaluate one's (and other people's) competence in that domain. *Metacognition* is the term used to denote the ability of a person to accurately judge how well they are performing.

Peoples' belief in their own approach to getting things done can result in them ignoring higher performing alternatives;[67] this behavior has become known as *the illusion of control*.[1070]

Studies[1319] have found cultural and context dependent factors influencing overconfidence; see Github–developers/journal-pone-0202288.R.

It might be thought that people, who have previously performed some operation, would be in a position to make accurate predictions about future performance on these operations. However, studies have found[1591] that, while people do use their memories of the duration of past events to make predictions of future event duration, their memories are systematic underestimates of past duration. People appear to underestimate future event duration because they underestimate past event duration.

2.8.6 Time discounting

People seek to consume pleasurable experiences sooner, and to delay their appointment with painful experiences, i.e., people tend to accept less satisfaction in the short-term, than could be obtained by pursuing a longer-term course of action; a variety of models have been proposed.[502] Studies have found that animals, including humans, appear to use a hyperbolic discount function for time variable preferences.[473] The hyperbolic delay discount function is:

$v_d = \frac{V}{1+kd}$, where: v_d is the delayed discount, V the undiscounted value, d the delay and k some constant.

A property of this hyperbolic function is that curves with different values of V and d can cross. Figure 2.67 shows an example where the perceived present value of two future rewards (red and blue lines) starts with red have a perceived value greater than blue, as time passes (i.e., the present time moves right, and the delay before receiving the rewards decreases) the perceived reward denoted by the blue line out-grows the red line, until there is a reversal in perceived present value. When both rewards are far in the future, the larger amount has the greater perceived value; studies have found[997] that subjects switch to giving a higher value to the lesser amount as the time of receiving the reward gets closer.

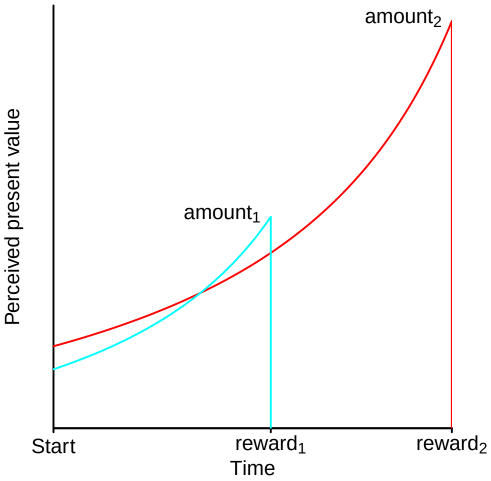


Figure 2.67: Perceived present value (moving through time to the right) of two future rewards. Github–Local

A study by Becker, Fagerholm, Mohanani and Chatzigeorgiou[154] asked professional developers how many days of effort would need to be saved, over the lifetime of a project, to make it worthwhile investing time integrating a new library, compared to spending that time implementing a feature in the project backlog. The developers worked at companies who developed and supported products over many years, and were asked to specify saving for various project lifetimes.

Figure 2.68 shows the normalised savings specified by developers from one company, for given project lifetimes. Many of the lines are concave, showing that these developers are applying an interest rate that decreases, for the time invested, as project lifetime increases (if a fixed interest rate, or hyperbolic rate, was applied, the lines would be convex, i.e., curve up).

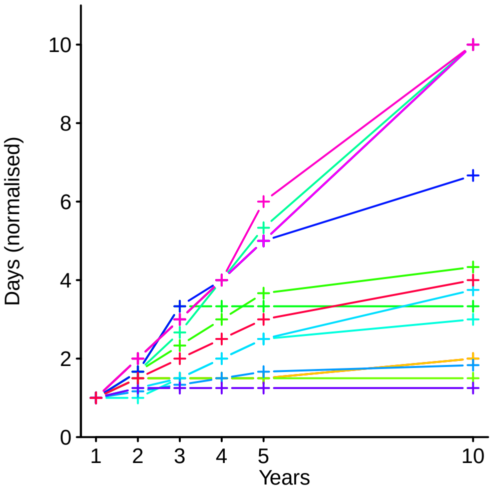


Figure 2.68: Saving required (normalised), over a project having a given duration, before subjects would make a long term investment. Data from Becker et al.[154] Github–Local

2.8.7 Developer performance

Companies seek to hire those people who will give the best software development performance. Currently, the only reliable method of evaluating developer performance is by measuring developer outputs (this is a good enough model of the workings of human mental operations remains in the future). Conscientiousness has consistently been found to be an effective predictor of occupational performance.[1949]

One operational characteristic of the brain that can be estimated is the number of operations that could potentially be performed per second (a commonly used method of estimating the performance of silicon-based processors).

The brain might simply be a very large neural net, so there may be no instructions to count as such; Merkle[1250] used the following approaches to estimate the number of synaptic operations per second (the supply of energy needed to fire neurons limits the number that can be simultaneously active, in a local region, to between 1% and 4% of the neurons in that region[1102]):

- multiplying the number of synapses (10^{15}), by their speed of operation (about 10 impulses/second), gives 10^{16} synapse operations per second (if the necessary energy could be delivered to all of them at the same time),
- the retina of the eye performs an estimated 10^{10} analog add operations per second. The brain contains 10^2 to 10^4 times as many nerve cells as the retina, suggesting that it can perform 10^{12} to 10^{14} operations per second,
- the brain's total power dissipation of 25 watts (an estimated 10 watts of useful work), and an estimated energy consumption of $5 \cdot 10^{-15}$ joules for the switching of a nerve cell membrane, provides an upper limit of $2 \cdot 10^{15}$ operations per second.

A synapse switching on and off is the same as a transistor switching on and off, in that they both need to be connected to other switches to create a larger functional unit. It is not known how many synapses are used to create functional units, or even what those functional units might be. The distance between synapses is approximately 1 mm, and sending a signal from one part of the brain to another part requires many synaptic operations, for instance, to travel from the front to the rear of the brain requires at least 100 synaptic operations to propagate the signal. So the number of synaptic operations per high-level, functional operation, is likely to be high. Silicon-based processors can contain millions of transistors; the potential number of transistor-switching operations per second might be greater than 10^{14}, but the number of instructions executed is significantly smaller.

Although there have been studies of the information-processing capacity of the brain (e.g., visual attention,[1871] and storage rate into long-term memory[234, 1063]), we are a long way from being able to deduce the likely work rates of the components of the brain while performing higher level cognitive functions.

Processing units need a continuous supply of energy to function. The brain does not contain any tissue that stores energy, and obtains all its energy needs through the breakdown of blood-borne glucose. Consuming a glucose drink has been found to increase blood glucose levels, and enhance performance on various cognitive tasks.[975] Fluctuations in glucose levels have an impact on an individual's ability to exert self-control,[634] with some glucose intolerant individuals not always acting in socially acceptable ways.[xiv]

How do developers differ in their measurable output performance?

Although much talked about, there has been little research on individual developer productivity (a few studies[1616] have used project level data to estimate productivity). One review[868] of studies of employee output variability, found that standard deviation, about the mean, increased from 19% to 48%, as job complexity increased (not software related). Claims of a 28-to-1 productivity difference between developers, is sometimes still bandied about. The, so-called *Grant-Sackman study*[719] is based on an incorrect interpretation of a summary of their experimental data.[1502] The data shows a performance difference of around 6-to-1 between developers using batch vs. online, for creating software; see Github–group-compare/GS-perm-diff.R and fig 8.22.

Lines of working code produced per unit-time is sometimes used; figures in the hundreds of instructions per man-month can sometimes be traced back to measurements made in the 1960s.[793]

A study by Nichols[1357] investigated the performance of those attending the Personal Software Process (PSP) training given at CMU (average professional experience 3.7 years, sd 6.5). The training involves writing programs to solve 10 problems, with each expected to take a few hours; participants also have to estimate how long various components of the task will take and record actual times.

Figure 2.69 shows violin plots for the actual time taken by the 593 participants, programming in C, to solve the problems (sorted by the mean solution time; white line shows mean time, colors intended to help visualise individual plots). There is almost an order of magnitude difference between developers, and between individual performance on different problems.

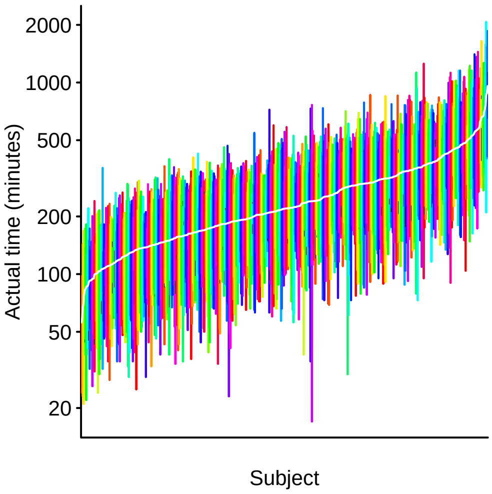


Figure 2.69: Violin plots for actual time to complete problems for each of the 593 participants, sorted by mean solution time; colors to help break up the plots, and white line shows subject mean. Data from Nichols.[1357] Github–Local

Most organizations do not attempt to measure the mental performance of job applicants for developer roles; unlike many other jobs where individual performance is an important consideration. Whether this is because of existing non-measurement culture, lack of reliable measuring procedures, or fear of frightening off prospective employees is not known.

One study[1981] of development and maintenance costs of programs written in C and Ada found no correlation between salary grade (or employee rating), and rate of bug fix or feature implementation rate.

One metric used in software testing is number of fault experiences encountered. In practice non-failing tests, written by software testers, are useful because they provide evidence that particular functionality behaves as expected.

A study by Iivonen[874] analysed the defect detection performance of those involved in testing software at several companies. Table 2.8 shows the number of defects detected by six testers (all but the first column, show percentages), along with self-classification of seriousness, followed by the default status assigned by others.

A tester performance comparison, based on defects reported, requires combining these figures (and perhaps others, e.g., likelihood of fault being experienced by a customer) into a value that can be reliably compared across testers. Defects differ in their commercial importance, and a relative weight for each classification has to be decided, e.g., should the weight of "No fix" be larger than that given to "Cannot reproduce" or "Duplicate"?

[xiv] There is a belief in software development circles that consumption of chocolate enhances cognitive function. A review of published studies[1627] found around a dozen papers addressing this topic, with three finding some cognitive benefits and five finding some improvement in mood state.

Tester	Defects	Extra Hot	Hot	Normal	Open	Fixed	No fix	Duplicate	Cannot reproduce
A	74	4	1	95	12	62	26	12	0
B	73	0	56	44	15	87	6	2	5
C	70	0	29	71	36	71	24	0	4
D	51	0	27	73	33	85	6	0	9
E	50	2	16	82	30	89	9	0	3
F	18	0	22	78	22	64	14	0	21

Table 2.8: Defects detected by six testers (left two columns; some part-time and one who left the company during the study period), the percentage assigned a given status (next three columns), and percentage outcomes assigned by others. Data from Iivonen.[874]

To what extent would a tester's performance, based on measurements involving one software system in one company, be transferable to another system in the same company or another company? Iivonen interviewed those involved in testing, to find out what characteristics were thought important in a tester. Knowledge of customer processes and use cases, was a common answer; this customer usage knowledge enables testers to concentrate on those parts of the software that customers are most likely to use and be impacted by incorrect operation, it also provides the information needed to narrow down the space of possible input values.

Knowledge of the customer ecosystem and software development skills are the two blades, in figure 2.1, that have to mesh together to create useful software systems.

2.8.8 Miscellaneous

Research in human-computer interaction has uncovered a variety of human performance characteristics, including: physical movement (e.g., hand or eye movement) and mental operations. The equations fitted to experimental performance data for some of these characteristics often contain logarithms, and attention has been drawn to the similarity of form to the equations used in information theory.[1174] The use of a logarithmic scale, to quantify the perception of stimuli, minimizes relative error.[1490] Some commonly encountered performance characteristics include:

- Fitts' law: time taken, RT, to move a distance D, to an object having width W, is: $RT = a + b\log\left[\frac{2D}{W}\right]$, where a and b are constants. In deriving this relationship, Fitts drew on ideas from information theory, and used a simplified version of Shannon's law; the unsimplified version implies: $RT = c + d\log\left[\frac{D+W}{W}\right]$,[1174]
- Hick's law: time taken, RT, to choose an item from a list of K items, is: $RT = a + b\log(K)$, where a and b are constants; a is smaller for humans than pigeons.[1879] A study by Hawkins, Brown, Steyvers and Wagenmakers[783] displayed a number of squares on a screen, and asked subjects to select the square whose contents had a particular characteristic. Figure 2.70 shows how subject response time increased (and accuracy decreased), as the log of number of choices. A different color is used for each of the 36 subjects, with colors sorted by performance on the two-choice case; data has been jittered to help show the density of points. This study found that problem context could cause subjects to make different time/accuracy performance trade-offs,

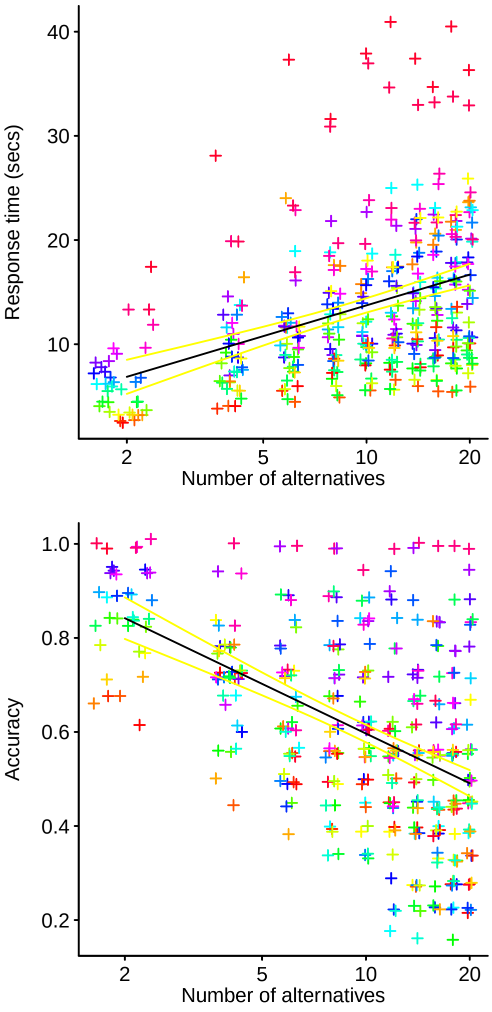


Figure 2.70: Mean time for each of 36 subjects to choose between a given number of alternatives (upper), and accuracy rate for a given number of alternatives (lower), data has been jittered; lines are regression fits (yellow shows 95% confidence intervals), and color used for each subject sorted by performance on the two-choice case. Data from Hawkins et al.[783] Github–Local

- Ageing effects: the Seattle Longitudinal Study[1620] has been following the intellectual development of over six thousand people since 1956 (surveys of individuals in the study are carried out every seven years). Findings include: " . . . there is no uniform pattern of age-related changes across all intellectual abilities, . . . ", and " . . . reliable replicable average age decrements in psychometric abilities do not occur prior to age 60, but that such reliable decrements can be found for all abilities by age 74 . . . " An analysis[221] of the workers on the production line at a Mercedes-Benz assembly plant found that productivity did not decline until at least up to age 60.

  Studies of professional developers that have included age related information,[920,923] have not found an interaction with subject experimental performance. See figure 8.13 for an example of developer age distribution,
- the *sunk cost effect* is the tendency to persist in an endeavour, once an investment of time, or money, has been made.[202] This effect is observed in other animals,[1338] and so presumably this behavior provides survival benefits.

Chapter 3

Cognitive capitalism

3.1 Introduction

Software systems are intangible goods that are products of cognitive capitalism; human cognition is the means of production.

Intangible goods can incur intangible costs during their production, such as the emotional distress experienced by developers when their ideas or work goes unused or unnoticed.

Major motivations for an individual to be involved in the production of software include money and hedonism. People might be motivated to write software by the salary they are paid, by the owners of an organization that employs them, or they might be motivated by non-salary income (of an individual's choosing), e.g., enjoyment from working on software systems, scratching an itch, being involved in a social activity, etc.

Motivation may be affected by the work environment in which software is produced. To make the cognitariate feel happy and fulfilled, spending their cognitive capital striving to achieve management goals, companies have industrialised Bohemia.

While salary based production is likely to be distributed under a commercial license, and hedonism based production under some form of Open source[i] license, this is not always the case.

This chapter discusses cognitive capitalism using the tools of economic production, which deals with tradeable quantities, such as money, time and pleasure. Many of the existing capitalist structures are oriented towards the production of tangible goods,[ii] and are slowly being adapted to deal with the growing market share of intangible goods.[171, 775]

This book is oriented towards the producers of software, rather than its consumers, e.g., it focuses on maximizing the return on investment for producers of software.

Human characteristics that affect cognitive performance are the subject of chapter 2.

The sector economic model groups human commercial activities into at least three sectors:[974] the primary sector produces or extracts raw materials, e.g., agriculture, fishing and mining, the secondary sector processes raw materials, e.g., manufacturing and construction, and the tertiary sector provides services. The production of intellectual capital is sometimes referred to as the quarternary sector. Figure 3.1 shows the percentage of the US workforce employed in the three sectors over the last 160 years (plus government employment).

How much money is spent on software production?

A study by Parker and Grimm[1425] investigated business and government expenditure on software in the U.S.A. They divided software expenditure into two categories: *custom software*, as software tailored to the specifications of an organization, and *own-account software* which consists of in-house expenditures for new or significantly-enhanced software created by organizations for their own use. Figure 3.2 shows annual expenditure from 1959 to 1998, by US businesses (plus lines), and the US federal and state governments (smooth lines).

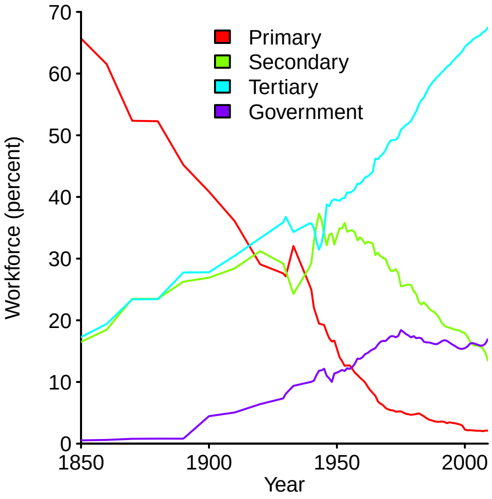


Figure 3.1: Percentage of employment by US industry sector 1850-2009. Data kindly provided by Kossik.[1026] Github–Local

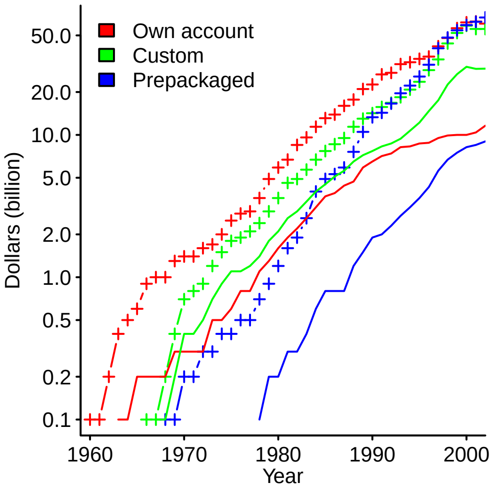


Figure 3.2: Annual expenditure on custom, own account and prepackaged software by US business (plus lines) and the US federal and state governments (smooth lines). Data from Parker et al.[1425] Github–Local

[i] The term is used generically, to refer to any software where the source code is freely available under a non-commercial license.

[ii] High-tech trade conflicts traditionally involved disparities in the quantity of hardware items being imported/exported annually.[1838]

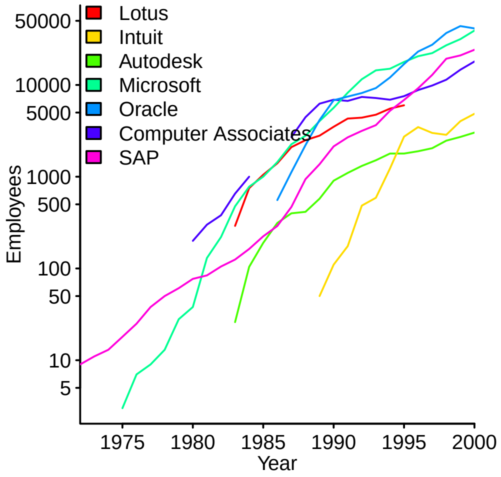


Figure 3.3: Number of people employed by major software companies. Data from Campbell-Kelly.[289] Github–Local

Figure 3.3 shows the growth in the number of people employed by some major software companies.

Some governments have recognized the importance of national software ecosystems,[1384] both in economic terms (e.g., industry investment in software systems[339] that keep them competitive), and as a means of self-determination (i.e., not having important infrastructure dependent on companies based in other countries); there is no shortage of recommendations[1771] for how to nurture IT-based businesses, and government funded reviews of their national software business.[1181, 1578] Several emerging economies have created sizeable software industries.[74]

The software export figures given for a country can be skewed by a range of factors, and the headline figures may not include associated costs.[791]

What percentage of their income do software companies spend on developing software? A study by Mulford and Misra[1306] of 100 companies in Standard Industry Classifications (SIC) 7371 and 7372[iii], with revenues exceeding $100 million during 2014–2015, found that total software development costs were around 19% of revenue; see figure 3.4; sales and marketing varies from 22% to 40%,[327] general and administrative (e.g., salaries, rent, etc) varies from 11% to 22%,[327] with the any remainder assigned to profit and associated taxes.

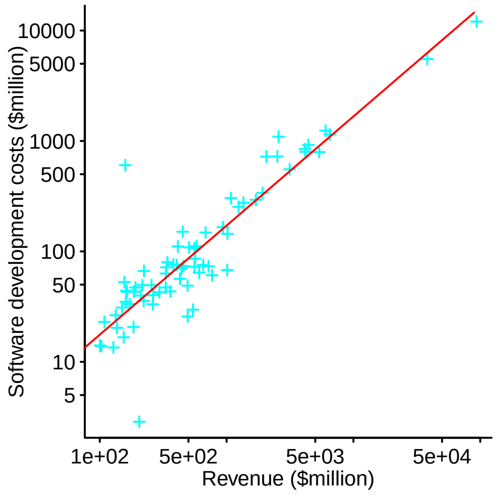


Figure 3.4: Company revenue ($millions) against total software development costs; line is a fitted regression model of the form: $developmentCosts \propto 0.19 Revenue$. Data from Mulford et al.[1306] Github–Local

3.2 Investment decisions

Creating software is an irreversible investment, i.e., incomplete software has little or no resale value, and even completed software may not have any resale value.

The investment decisions involved in building software systems share some of the characteristics of other kinds of investments; for instance, making sequential investments, with a maximum construction rate, are characteristics that occur in factory construction.[1180]

Is it worthwhile investing resources, to implement software that provides some desired functionality? In the case of a personal project, an individual may decide on the spur of the moment to spend time implementing or modifying a program; for commercial projects a significant early stage investment may be made analyzing the potential market, performing a detailed cost/benefit analysis, and weighing the various risk factors.

This sections outlines the major factors involved in making investment decisions, and some of the analysis techniques that may be used. While a variety of sophisticated techniques are available, managers may choose to use the simpler ones.[1789]

The term *cost/benefit* applies when making a decision about whether to invest or not; the term *cost-effectiveness* applies when a resource is available, and has to be used wisely, or when an objective has to be achieved as cheaply as possible.

Basic considerations for all investment decisions include:

- the value of money over time. A pound/dollar in the hand today is worth more than a pound/dollar in the hand tomorrow,
- risks. Investors require a greater return from a high risk investment, than from a low risk investment,
- uncertainty. The future is uncertain, and information about the present contains uncertainty,
- *opportunity cost*. Investing resources in project *A* today, removes the opportunity to invest them project *B* tomorrow. When investment opportunities are abundant, it is worth searching for one of the better ones, while when opportunities are scarce searching for alternatives may be counter-productive.

  For instance, when deciding between paying for Amazon spot instances[168] at a rate similar to everybody else, or investing time trying to figure out an algorithm that makes it possible to successfully bid at much lower prices, the time spent figuring out the algorithm is an opportunity cost (i.e., the time spent is a lost opportunity for doing something else, which may have been more profitable).[iv]

Return on investment (ROI) is defined as:

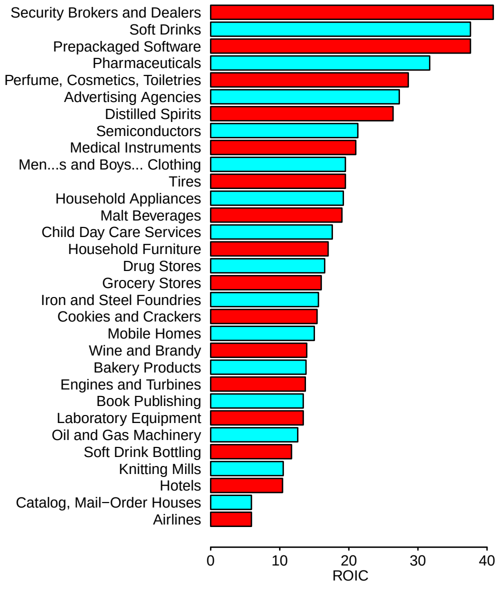


Figure 3.5: Average Return On Invested Capital of various U.S. industries between 1992-2006. Data from Porter.[1489] Github–Local

[iii] Computer programming services and Prepackaged software.

[iv] There is also the risk that the result of successfully reverse engineering the pricing algorithm results in Amazon changing the algorithm.[168]

$R_{est} = \frac{B_{est} - C_{est}}{C_{est}}$, where: B_{est} is the estimated benefit, and C_{est} the estimated cost.

Both the cost, and the benefit estimates are likely to contain some amount of uncertainty, and the minimum and maximum ROI are given by:

$$R_{est} - \delta R = \frac{B_{est} - \delta B}{C_{est} + \delta C} - 1 \quad \text{and} \quad R_{est} + \delta R = \frac{B_{est} + \delta B}{C_{est} - \delta C} - 1$$

where: δ is the uncertainty in the corresponding variable.

In practice, ROI uncertainty is unlikely to take an extreme value, and its expected value is given by:[224]

$$E[\delta R] \approx \frac{B_{est}}{C_{est}} \sqrt{\left(\frac{\delta B}{B_{est}}\right)^2 + \left(\frac{\delta C}{C_{est}}\right)^2}$$

Figure 3.6 shows the development cost of video games (where the cost was more than $50million). The high risk of a market that requires a large upfront investment, to create a new product for an uncertain return, is offset by the possibility of a high return.

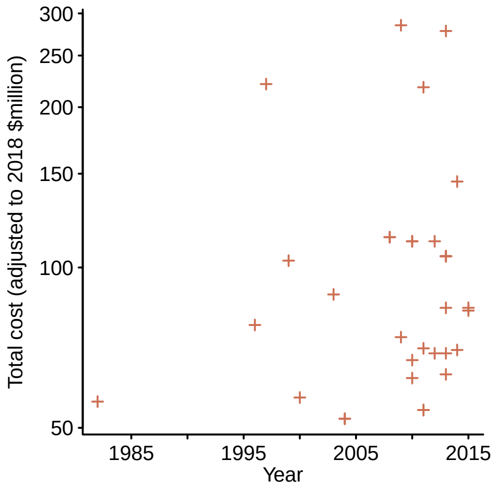


Figure 3.6: Development cost (adjusted to 2018 dollars) of computer video games, whose cost was more than $50million. Data from Wikipedia.[1937] Github–Local

3.2.1 Discounting for time

A dollar today is worth more than a dollar tomorrow, because today's dollar can be invested and earn interest; by tomorrow, the amount could have increased in value (or at least not lost value, through inflation). The present value (PV) of a future payoff, C, might be calculated as:

$PV = discount\_factor \times C$, where: $discount\_factor < 1$.

discount_factor is usually calculated as: $(1+r)^{-1}$, where: r is the known as the *rate of return* (also known as the *discount rate*, or the *opportunity cost* of capital), and represents the size of the reward demanded by investors for accepting a delayed payment (it is often quoted for a period of one year).

The PV over n years (or whatever period r is expressed in) is given by:

$$PV = \frac{C}{(1+r)^n}$$

When comparing multiple options, expressing each of their costs in terms of present value enables them to be compared on an equal footing.

For example, consider the choice between spending $250,000 purchasing a test tool, or the same amount on hiring testers; assuming the tool will make an immediate cost saving of $500,000 (by automating various test procedures), while hiring testers will result in a saving of $750,000 in two years time. Which is the more cost-effective investment (assuming a 10% discount rate)?

$$PV_{tool} = \frac{\$500,000}{(1+0.10)^0} = \$500,000 \qquad \text{and} \qquad PV_{testers} = \frac{\$750,000}{(1+0.10)^2} = \$619,835$$

Based on these calculations, hiring the testers is more cost-effective, i.e., it has the greater present value.

3.2.2 Taking risk into account

The calculation in the previous section assumed there was no risk of unplanned events. What if the tool did not perform as expected, what if some testers were not as productive as hoped? A more realistic calculation of present value would take into account the possibility that future payoffs are smaller than expected.

A risky future payoff is worth less than a certain future payoff, for the same amount invested and payoff. Risk can be factored into the discount rate, to create an *effective discount rate*: $k = r + \theta$, where: r is the risk-free rate, and θ a premium that depends on the amount of risk. The formulae for present value becomes:

$$PV = \frac{C}{(1+k)^n}$$

When r and θ can vary over time, we get:

$PV = \sum_{i=1}^{t} \frac{return_i}{(1+k_i)^i}$, where: $return_i$ is the return during period i.

Repeating the preceding example, assuming a 15% risk premium for the testers option, we get:

$$PV_{tool} = \frac{\$500,000}{(1+0.10)^0} = \$500,000 \quad \text{and} \quad PV_{testers} = \frac{\$750,000}{(1+0.10+0.15)^2} = \$480,000$$

Taking an estimated risk into account, suggests that buying the tool is the most cost-effective of the two options.

The previous analysis compares the two benefits, but not the cost of the investment that needs to be made to achieve the respective benefit. Comparing investment costs requires taking into account when the money is spent, to calculate the total cost terms of a present cost.

Purchasing the tool is a one time, up front, payment:

$investment\_cost_{tool} = \$250,000$

The cost of the testers approach is more complicated; let's assume it is dominated by monthly salary costs. If the testing cost is $10,416.67 per month for 24 months, the total cost after two years, in today's terms, is (a 10% annual interest rate is approximately 0.8% per month):

$$investment\_cost_{testers} = \sum_{m=0}^{23} \frac{\$10,416.67}{(1+0.008)^m} = \$10,416.67 \left[\frac{1-(1+0.008)^{-22}}{1-(1+0.008)^{-1}} \right] = \$211,042.90$$

Spending $250,000 over two years is equivalent to spending $211,042.90 today. Investing $211,042.90 today, at 10%, would provide a fund that supports spending $10,416.67 per month for 24 months.

Net Present Value (NPV) is defined as: $NPV = PV - investment\_cost$

Plugging in the calculated values gives:

$NPV_{tool} = \$500,000 - \$250,000 = \$250,000$

$NPV_{testers} = \$480,000 - \$211,042.90 = \$268,957.10$

Based on NPV, hiring testers is the more cost-effective option.

Alternatives to NPV, their advantages and disadvantages, are discussed by Brealey[245] and Raffo.[1532] One commonly encountered rule, in rapidly changing environments, is the payback rule, which requires that the investment costs of a project be recovered within a specified period; the *payback period* is the amount of time needed to recover investment costs (a shorter payback period being preferred to a longer one).

Accurate estimates for the NPV of different options requires accurate estimates for the discount rate, and the impact of risk. The discount rate represents the risk-free element, and the closest thing to a risk-free investment is government bonds and securities (information on these rates is freely available). Governments face something of a circularity problem in how they calculate the discount rate for their own investments. The US government discusses these issues in its "Guidelines and Discount Rates for Benefit-Cost Analysis of Federal Programs",[1928] and at the time of writing the revised Appendix C specified rates, varied between 0.9% over a three-year period and 2.7% over 30 years. Commercial companies invariably have to pay higher rates of interest than the US Government.

3.2.3 Incremental investments and returns

Investments and returns are sometimes incremental, occurring at multiple points in time, over an extended period.

The following example is based on the idea that it is possible to make an investment, when writing or maintaining code, that reduces subsequent maintenance costs, i.e., produces a return on the investment. At a minimum, any investment made to reduce later maintenance costs must be recouped; this minimum case has an ROI of 0%.

Let d be the original development cost, m the base maintenance cost during time period t, and r the interest rate; to keep things simple assume that m is the same for every period of maintenance; the NPV for the payments over t periods is:

$$Total\_cost = d + \sum_{k=1}^{t} \frac{m}{(1+r)^k} = d + m\left[\frac{1-(1+r)^{-(t+1)}}{1-(1+r)^{-1}} - 1\right] \approx d + m\times t\,[1-0.5\times(t+1)r]$$

with the approximation applying when $r \times t$ is small.

If an investment is made for all implementation work (i.e., development cost is: $(1+i)d$), in expectation of achieving a reduction in maintenance costs during each period of $(1-b)$, then:

$$Total\_cost_{invest} = (1+i)d + m\times(1+i)(1-b)\times t\,[1-0.5\times(t+1)r]$$

For this investment to be worthwhile: $Total\_cost_{invest} \leq Total\_cost$, giving:

$(1+i)d + m\times(1+i)(1-b)\times T < d + m\times T$, where: $T = t\,[1-0.5\times(t+1)r]$, which simplifies to give the following lower bound for the benefit/investment ratio, $\frac{b}{i}$:

$$1 + \frac{d}{mT} < \frac{b}{i} + b$$

In practice many systems have a short lifetime. What value must the ratio $\frac{b}{i}$ have, for an investment to break even, after taking into account system survival rate (i.e., the possibility that there is no future system to maintain)?

If s is the probability the system survives a maintenance period, the total system cost is:

$$Total\_cost = d + \sum_{k=1}^{t} \frac{m\times s^k}{(1+r)^k} = d + m\frac{S(1-S^t)}{1-S}, \text{ where: } S = \frac{s}{1+r}.$$

The minimal requirement is now:

$$1 + \frac{d}{m}\frac{1-S}{S(1-S^t)} < \frac{b}{i} + b$$

The development/maintenance break-even ratio depends on the regular maintenance cost, multiplied by a factor that depends on the system survival rate (not the approximate total maintenance cost).

Fitting regression models to system lifespan data, in section 4.5.2, finds (for an annual maintenance period): $s_{mainframe} = 0.87$ and $s_{Google} = 0.79$. Information on possible development/maintenance ratios is only available for mainframe software (see fig 4.47), and the annual mean value is: $\frac{d}{m} = 4.9$.

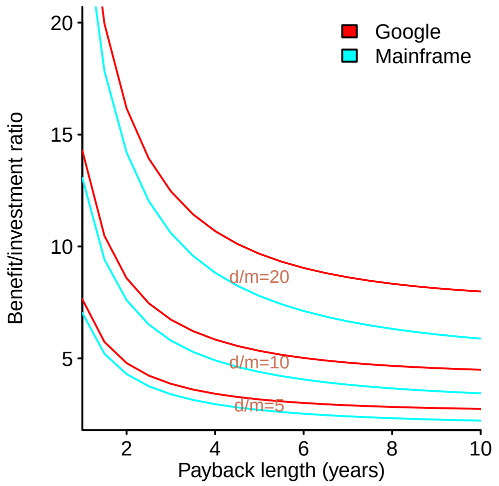


Figure 3.7: Return/investment ratio needed to break-even, for *Google* and *Mainframe* application survival rate, having development/annual maintenance ratios of 5, 10 and 20; against payback period in years. Data from: mainframe Tamai,[1791] Google SaaS Ogden.[1386] Github–Local

Figure 3.7 shows the multiple of any additional investment (y-axis), made during development, that needs to be recouped during subsequent maintenance work, to break even for a range of payback periods (when application survival rate is that experienced by Google applications, or 1990s Japanese mainframe software; the initial development/annual maintenance ratios are 5, 10 and 20); the interest rate used is 5%.

This analysis only considers systems that have been delivered and deployed. Projects are sometimes cancelled before reaching this stage, and including these in the analysis would increase the benefit/investment break-even ratio.

While some percentage of a program's features may not be used,[522] those features that will go unused is unknown at the time of implementation. Incremental implementation, driven by customer feedback, helps reduce the likelihood of investing in program features that are not used by customers.

Figure 7.16 shows that most files are only ever edited by one person. The probability that source will be maintained by developers other than the original author may need to be factored into the model.

3.2.4 Investment under uncertainty

The future is uncertain, and the analysis in the previous sections assumed fixed values for future rates of interest and risk. A more realistic analysis would take into account future uncertainty.[496, 861]

An investment might be split into random and non-random components, e.g., Brownian motion with drift. The following equation is used extensively to model the uncertainty involved in investment decisions[496] (it is a stochastic differential equation for the change in the quantity of interest, x):

$dx = a(x,t)dt + b(x,t)dz$

This equation contains a drift term (given by the function a, which involves x and time) over an increment of time, dt, plus an increment of a Wiener process,[v] dz (the random component; also known as a Brownian process), and the function, b, involves x and time.

The simplest form of this equation is: $dx = \alpha dt + \sigma dz$, i.e., a constant drift rate, α, plus a random component having variance σ. Figure 3.8 illustrates this drift-diffusion process; with the grey line showing the slope of the drift component, and green lines showing possible paths followed when random increments are added to the drift (red lines bound possible paths, for the value of σ used).

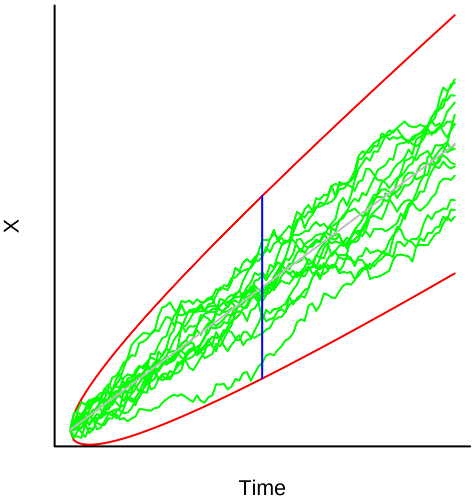


Figure 3.8: Illustration of a drift diffusion process. Green lines show possible paths, red lines show bounds of diffusion and grey line shows drift with no diffusion component. Github–Local

This equation can be solved to find the standard deviation in the value of x: it is $\sigma\sqrt{T}$, where T is elapsed time.

The analysis of the cost/benefit of the maintenance investment, in the previous section, assumed a fixed amount of maintenance in each interval. In practice, the amount of maintenance is likely to grow to a maximum commercially supportable value, and then fluctuate about this value over time, i.e., it becomes a mean-reverting process. The *Ornstein-Uhlenbeck process* (also known as the *Vasicek process*) is the simplest mean-reverting process, and its corresponding equation is:

$$dx = \eta(\hat{x} - x)dt + \sigma dz \tag{3.1}$$

where: η is the speed of reversion, and $\hat{x}$ is the mean value.

This equation can be solved to give the expected value of x at future time t, given its current value x_0, which is: $E[x_t] = \hat{x} + (x_0 - \hat{x})e^{-\eta t}$; its variance is: $\frac{\sigma^2}{2\eta}(1 - e^{-2\eta t})$; section 11.10.7 discusses techniques for obtaining values for σ and η.

Figure 3.9 shows ten example paths (in green) of an Ornstein-Uhlenbeck process, each starting at zero, and growing to fluctuate around the process mean; red line is the expected value, blue lines are at one standard deviation.

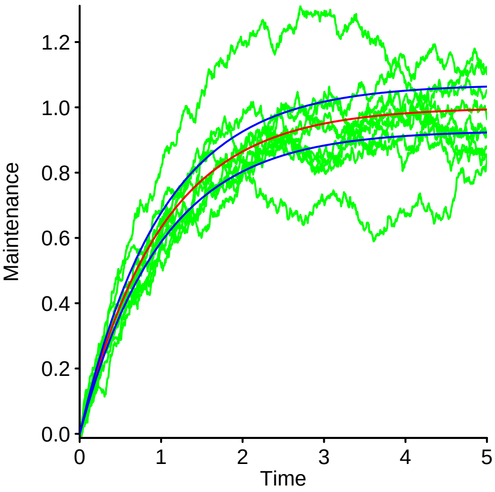


Figure 3.9: Illustration of an Ornstein-Uhlenbeck process starting from zero and growing to its mean; green lines show various possible paths, red line is expected value, and blue lines one standard deviation. Github–Local

Uncertainty in the investment cost and/or the likely benefit returned from a project, means that even under relatively mild conditions, a return of twice as much as the investment is considered to be the break-even condition.[496, 1468]

Obtaining insight about a possible investment, by setting up and solving a stochastic differential equation[1637] can be very difficult, or impractical. A variety of numerical methods are available for analyzing investment problems based on a stochastic approach, see section 11.10.7.

Current market conditions also need to be taken into account. For instance: how likely is it that other companies will bring out competing products, and will demand for the application still be there once development is complete?

3.2.5 Real options

It may be possible to delay making an investment until circumstances are more favorable.[392, 496] For instance, when new functionality has to be implemented as soon as possible, it may be decided to delay investing the resources needed to ensure the new code conforms to project guidelines; there is always the option to invest the necessary resources later. By using Net Present Value, management is taking a passive approach to their investment strategy; a fixed calculation is made, and subsequent decisions are based on this result; real options offer a more flexible approach.

In financial markets, a *call option* is a contract between two parties (a buyer and a seller), where the buyer pays the seller for the right (but not the obligation) to buy (from the seller) an asset, at an agreed price on an agreed date (the initial amount paid is known as the *premium*, the agreed price the *strike price*, and the date the *expiry* or *maturity* date). When the option can only be exercised on the agreed date, the contract is known as a *European option*, when the option can be exercised at any time up to the agreed date, it is known as a *American option*.

A *put option* is a contract involving the right (but not the obligation) to sell.

[v] A Wiener process involves random incremental changes, with changes being independent of each other, and the size of the change having a Normal distribution.

The term *real options* (or *real options valuation* ROV, or *real options analysis*) is applied to the analysis of decisions made by managers in industry that have option-like characteristics. In this environment many items are not traded like conventional financial options; another notable difference, is that those involved in the management of the asset, may have an influence on the value of the option (e.g., they have a say in the execution of a project).

In financial markets, volatility information can be obtained from an assets past history. In industry, managers may have information on previous projects carried out within the company, but otherwise have to rely on their own judgment to estimate uncertainty. Call and put options are established practice in financial markets. In industry, managers have to create or discover possible options; they need an entrepreneurial frame of mind.

ROV requires active management, continuously ready to respond to changing circumstances. It is an approach that is most applicable when uncertainty is high, and managers have the flexibility needed to make the required changes.

The Binomial model[392] can be used to estimate the percentage change in starting costs S, at time t_i, given the probability p, of costs going up by $U\%$, and the probability $1-p$, of costs going down by $D\%$, at each time step.

Figure 3.10 illustrates how after three time-steps, there is one path where costs can increase by $U^3\%$, and one where they can decrease by increase by $D^3\%$; there are three paths where the cost can increase by $3U^2D\%$, and three where they can decrease by $3D^2U\%$. All paths leading to a given U/D point occur with the same probability, which can be calculated from p.

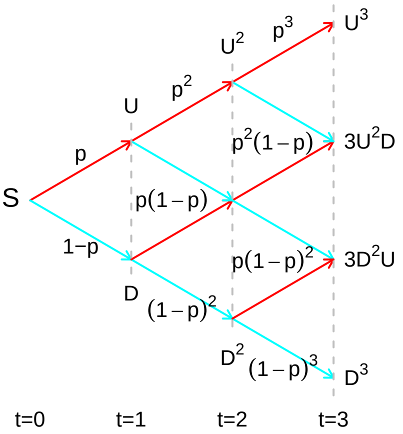


Figure 3.10: Example of a binomial model with three time-steps, given the probability p, of costs going up by $U\%$, and the probability $1-p$, of costs going down by $D\%$, at each time step, starting at S. Github–Local

Implementing functionality using the minimum of investment, with the intent of investing more later, has the form of an American call option.[vi] The call option might not be exercised, e.g., if the implemented functionality becomes unnecessary.

The Black-Scholes equation provides a solution to the optimal pricing of European options (e.g., the value of the premium, strike price, and maturity date). Some researchers have applied this equation to the options associated with developing software; this is a mistake.[588] The derivation of the Black-Scholes equation involves several preconditions, which often apply to financial risks, but don't apply to software development, including:

- liquidity is required, to enable the composition of a portfolio of assets to be changed, at will, by investing or divesting, i.e., buying or selling. Software production does not create liquid assets, the production is a sunk cost. Once an investment has been made in writing code, it is rarely possible to immediately divest a percentage of the investment in this code (a further investment in writing new code may make little business sense),
- detailed historical information on the performance of the item being traded is an essential input to portfolio risk calculations. Historical data is rarely available on one, let alone all, of the major performance factors involved in software development; chapter 5 discusses the distribution of development activities over the lifetime of a project.

Some of the issues involved in a cost/benefit analysis of finding and fixing coding mistakes is discussed in section 3.6.

3.3 Capturing cognitive output

Organizations whose income is predominantly derived from the output of cognitariate are social factories.[304] To increase the fraction of cognitive output they capture, organizations involve themselves in employees' lives to reduce external friction points; free meals and laundry service are not perks, they are a means of bringing employees together to learn and share ideas, by reducing opportunities to mix in non-employee social circles (and create employee life-norms that limit possible alternative employers, i.e., a means of reducing knowledge loss and spillover).

Organizations that continue to be based around work-life separation also use software systems, and offer software related jobs. In these organizations the primary focus may be on the cognitive efforts of people working on non-software activities, with those working on software related activities having to fit in with the existing workplace norms of the host industry.

[vi] A term in common use is *technical debt*; this is incorrect, there is no debt, and there may not be any need for more work in the future.

Some developers are more strongly motivated by the enjoyment from doing what they do, than the money they receive for doing it.[1310] This lifestyle choice relies on the willingness of organizations to tolerate workers who are less willing to do work they don't enjoy, or alternating between high paying work that is not enjoyed, and working for pleasure. Freelancing in tasks such as trying to earn bug bounties is only profitable for a few people (see fig 4.43).

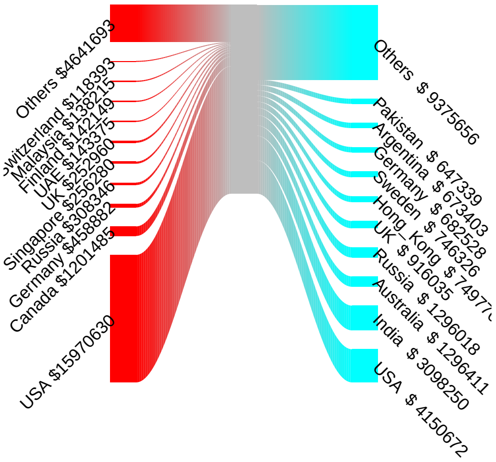


Figure 3.11: Bug bounty payer (left) and payee (right) countries (total value $23,632,408). Data from hackerone.[755] Github–Local

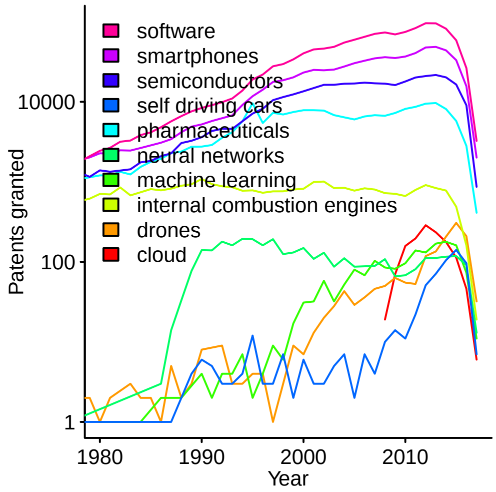


Figure 3.12: Number of US patents granted in various areas. Data from Webb et al.[1916] Github–Local

3.3.1 Intellectual property

Governments have created two legal structures that might be used by individuals, or organizations, to claim property rights over particular source code, or ways of doing things (e.g., business processes): patents and copyright.

A patent gives its owner rights to exclude others from making or selling an item that makes use of the claims made in the patent (for a term of 20 years from the date of filing; variations apply). In the U.S. software was not patentable until 1995, and since then the number of software patents has continued to grow, with 109,281 granted in 2014[1484] (36% of all utility patents). Figure 3.12 shows the number of US patents granted in various software related domains (the dip in the last few years is due to patents applications not yet granted).

Patents grant their holder exclusive use of the claimed invention in the jurisdiction that granted the patent. In a fast moving market the value of a patent may be in the strategic power[770] it gives to deny market access to competitive products. An analysis[761] of the stock-price of ICT companies in the US found that those with software patents had a slightly higher value than those without software patents.

A license is a legal document, chosen or created by the copyright owner of software, whose intended purpose is to control the use, distribution, and creation of derivative works of the software. Licensing is an integral component of the customer relationship.

Licensing is a source of ecological pressure (e.g., applications available under one kind of license may find it difficult to remain viable in ecosystems containing similar applications available under less restrictive licenses). The ideology associated with some kinds of licenses may be strong enough for people to create license-based ecosystems, e.g., Debian distributions only contain software whose source is licensed under a Free Software license. The restrictions imposed by a license, on the use of a software system, may be sufficiently at odds with the ideology of some developers that they decide to invest their time in creating a new implementation of the functionality under a different license.

Code distributed under an Open source license is often created using a non-contract form of production.[171]

People have been making source code available at no cost, for others to use, since software was first created;[68] however, the formal licensing of such software is relatively new.[1580] Open source licenses have evolved in response to user needs, court decisions, and the growth of new product ecosystems; there has been a proliferation of different, and sometimes incompatible, open source licenses.[1276] Factors driving the evolution of licensing conditions, and combinations of conditions, include: evolution of the legal landscape (e.g., issues around what constitutes distribution and derivative works[744]), commercial needs for a less restrictive license (e.g., the 3-clauses and 2-clauses variants of the original, more restrictive BSD license, now known as 4-clauses BSD), ideological demands for a more restrictive license (e.g., GPL[1039] version 3 added wording to address issues such as hardware locks and digital rights management, that were not addressed in GPL version 2), the desire to interoperate with software made available under an incompatible, but widely used, license (often the GPL),[663] or the desire to operate a business supporting a particular open source software system (e.g., dual licensing[1850]).

License selection, by commercial vendors, is driven by the desire to maximise the return on investment. Vendor licensing choice can vary from a license that ties software to a designated computer (with the expectation that the customer will later buy licenses for other computers), to a very permissive open source license[vii] (whose intent is to nullify a market entry-point for potential competitors).[291]

A study by Zhang, Yang, Lopes and Kim[1990] investigated Java code fragments, containing at least 50 tokens, appearing in answers to questions on Stack Overflow, that later

[vii] The term *permissive* is used to denote the extent to which an open source license, consistent with a particular ideological viewpoint, allows such licensed software to be used, modified and operated in conjunction with, other software licensed under a license consistent with different ideological viewpoint.

appeared within the source code of projects on Github. They found 629 instances of code on Github that acknowledged being derived from an answer on Stack Overflow, and 14,124 instances on Github of code that was similar to code in appearing Stack Overflow answers (a clone detection tool was used). The most common differences between the Stack Overflow answer and Github source were: use of a different method, and renaming of identifiers and types.

Figure 3.13 shows the normalised frequency of occurrences of matched code fragments containing a given number of lines, for the attributed use and unattributed close clone instances.

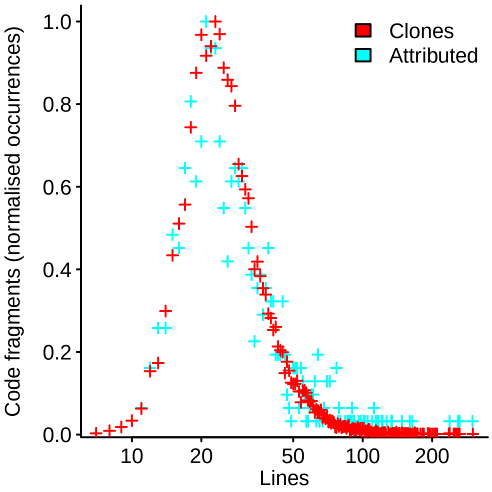


Figure 3.13: Normalised frequency of occurrences of code fragments containing a given number of lines; attributed to Stack Overflow answers, and unattributed close clones (a lognormal distribution is not sufficiently spikey to fit the data well). Data from Zhang et al.[1990] Github–Local

The majority of source code files do not contain an explicit license.[1870] Developers who do specify a license, may be driven by ideology, or the desire to fit in with the license choices made by others in their social network.[1694] One study[38] found that developers were able to correctly answer questions involving individual licenses (as-in, the answer agreed with a legal expert), but struggled with questions that required integrating conditions contained in multiple licenses.

Commercial vendors may invest in creating a new implementation of the functionality provided by software licensed under an Open source license, because, for instance, they want the licensing fee income, or the existing software has a license that does not suit their needs. For instance, Apple have been a longtime major funder of the LLVM compiler project, an alternative to GCC (one is licensed under the GPL, the other under the University of Illinois/NCSA open source license; the latter is a more permissive license).

Details of any applicable license(s) may appear within individual files, and/or in a license file within the top-level directory of the source code. In the same way that software can be changed, the license under which software is made available can change.[1886] Any changes to the licensing of a software system become visible to users when a new release is made.

When the files used to build a software system contain a license, different licenses may appear in different files, and the different licenses may specify conditions that are incompatible, with each other. For instance, Open source licenses might be classified as restrictive (i.e., if a modified version of the software is distributed, the derived version must be licensed under the same terms), or permissive (i.e., derived works may incorporate software licensed under different terms).

A study by Vendome, Linares-Vásquez, Bavota, Di Penta, German and Poshyvanyk[1870] investigated license use in the files of 16,221 Java projects (a later study also included projects written in other languages). Nearly all files did not include a license, and when a license was present it was rarely changed (the most common change, by an order of magnitude, was to/from a license being present).

Figure 3.14 shows the cumulative percentage of licenses present in files, over time, in the 10% of projects having the most commits. The lines show 16 of the 80 different licenses appearing in files, with the other 64 licenses appearing in less than 1,000 files. The downward trend, for some licenses, is caused by the increasing project growth rate, with most files not including any license (upper line in plot).

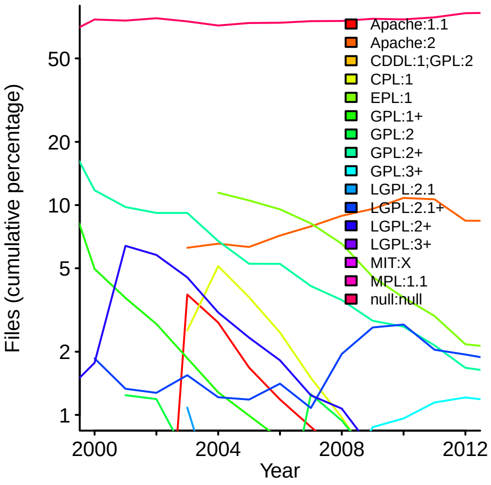


Figure 3.14: Cumulative percentage of files, from the top 10% largest Java projects, containing a given license (upper line is no license). Data from Vendome et al.[1870] Github–Local

A study by German, Manabe and Inoue[664] investigated the use of licenses in the source code of programs in the Debian distribution (version 5.0.2). They found that 68.5% of files contained some form of license statement (the median size of the license was 1,005 bytes). The median size of all files was 4,633 bytes; the median size of files without a license was 2,137 bytes, and the files with a license 5,488.

Programs and libraries may have dependencies on packages made available under licenses which specify conditions that are incompatible with each other, e.g., mobile apps having licenses that are incompatible with one or more of the licenses in the source files from which they were created.[1284]

A study by Meloca, Pinto, Baiser, Mattos, Polato, Wiese, and German[1244] investigated licensing files appearing in all packages from the `npm` (510,964 packages), `RubyGems` (135,481 packages), and `CRAN` (11,366 packages) networks (from the launch of the package site to October 2017).

Figure 3.15 shows the number of package releases containing a given number of licenses (some packages were updated, and an updated version released: there were 4,367,440 releases; a package must contain a license to appear on `CRAN`).

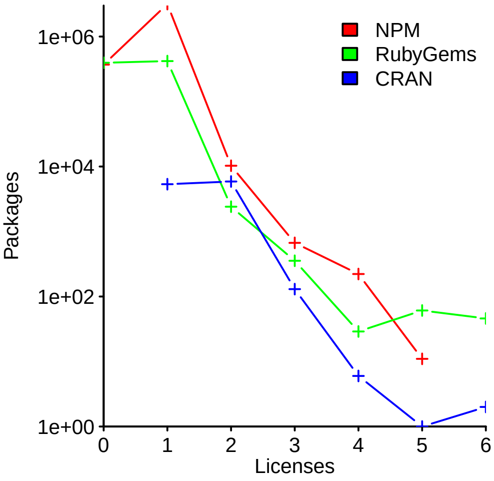


Figure 3.15: Number of releases of packages containing a given number of licenses (a package has to contain a license to appear on `CRAN`). Data from Meloca et al.[1244] Github–Local

Table 3.1 shows the percentage of all `npm` package releases using a given license, or lack thereof (column names), that migrated (or not) to another license (row names), from the launch of the npm network in 2009, until October 2017. During this period 85% of all package releases contained an OSI license; see Github–economics/msr2018b_evol.R.

To/From	OSI	Incomplete	non-OSI	Missing	Other	Copyright
OSI	99.7	6.8	7.7	2.6	4.8	2.8
Incomplete	0.12	92.0	0.1	0.09	3.3	3.0
non-OSI	0.08	0.07	91.1	0.07	0.95	0.81
Missing	0.08	0.22	0.65	97.2	0.49	0.26
Other	0.02	0.54	0.28	0.02	88.1	3.7
Copyright	0.00	0.31	0.2	0.01	2.3	90.0

Table 3.1: Percentage migration of licenses used by all npm package releases (between 2009 and 2017), from license listed in column names, to license in row names. Data from Meloca et al.[1244]

When licensing conditions are not followed, what action, if any, might the copyright holders of the software take?

The licensee has the option of instigating legal proceedings to protect their property. Those seeking to build market share may not take any action against some license violations (many people do not read end-user license agreements (EULAs)[1196]).

Some companies and individuals have sought to obtain licensing fees from users of some software systems made available under an Open source license. If the licensing fee sought is small enough, it may be cheaper for larger companies to pay, rather than legally contest the claims.[1924]

A license is interpreted within the framework of the laws of the country in which it is being used. For instance, the Uniform Commercial Code (UCC) is a series of laws, intended to harmonise commercial transactions across the US; conflicts between requirements specified in the UCC and a software license may have be resolved by existing case law, or by a new court case.[1227] Competing against free is very difficult. The Free Software Foundation were the (successful) defendants in a case where the plaintiff argued that the GPL violated the Sherman Act, e.g., “the restraint of trade by way of a licensing scheme to fix the prices of computer software products”.[1815]

A country’s courts have the final say in how the wording contained in software licenses is interpreted. The uncertainty over the enforceability[693, 1043] of conditions contained in Open source licenses is slowly being clarified, as more cases are being litigated, in various jurisdictions.[1136, 1829]

In litigation involving commercial licenses, the discovery that one party is making use of Open source software without complying with the terms of its Open source license, provides a bargaining chip for the other party,[1698] i.e., if the non-compliance became public, the cost of compliance, with the open source license, may be greater than the cost of the commercial licensing disagreement.

The Open Source Initiative (OSI) is a non-profit organization, that has defined what constitutes Open source, and maintains a list of licenses that meet this definition. OSI has become the major brand, and the largest organization operating in this area; licenses submitted, to become OSI-approved, have to pass a legal and community wide review.

Figure 3.16 shows the survival curve for the 107 OSI licenses that have been listed on the OSI website since August 2000; at the start of 2019, 81 licenses were listed on the OSI website.

A software system may be dependent on particular data, for correct or optimal operation. The licensing of data associated with software, or otherwise, can be an important issue itself.[789]

Open source ideology is being applied to other goods, in particular the hardware on which software runs.[217]

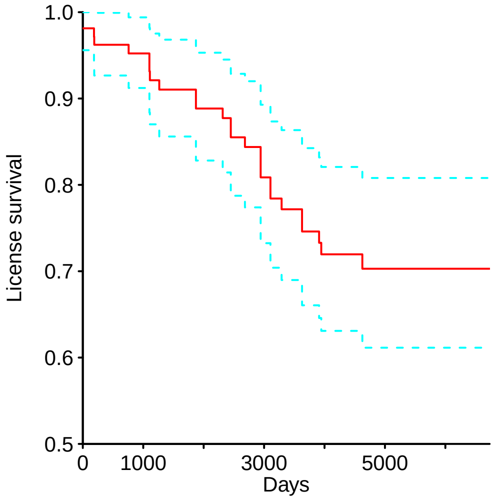


Figure 3.16: Survival curve of OSI licenses that have been listed on the approved license webpage, in days since 15 August 2000, with 95% confidence intervals. Data from opensource.org, via The Wayback Machine, web.archive.org. Github–Local

3.3.2 Bumbling through life

The expectation of career progression comes from an era when many industries were stable for long enough for career paths to become established. In rapidly changing work ecosystems, the concept of career progression is less likely to apply. Some software companies have established career progression ladders for their employees,[1729] and one study[1167] found that staff turnover was reduced when promotions were likely to be frequent and small, rather than infrequent and large; Gruber[743] discusses the issues of IT careers, in detail.

Companies have an interest in offering employees a variety of career options: it is a means of retaining the knowledge acquired by employees based on their experience of the business. Employees may decide that they prefer to continue focusing on technical issues, rather than people-issues, or may embrace a move to management. Simulations[1475] have found some truth to the Peter principle: "Every new member in a hierarchical organization climbs the hierarchy until they reach their level of maximum incompetence."

In fast changing knowledge-based industries, an individuals' existing skills can quickly become obsolete (this issue predates the computer industry[1887]). Options available to those with skills thought likely to become obsolete include, include having a job that looks like it will continue to require the learned skills for a many years, and investing in learning appropriate new skills.

Filtering out potential cognitariate who are not passionate about their work is a component of the hiring process at some companies. Passion is a double-edged sword; passionate employees are more easily exploited,[988] but they may exploit the company in pursuit of their personal ideals (e.g., spending time reworking code that they consider to be ugly, despite knowing the code has no future use).

Job satisfaction is an important consideration for all workers.[1310] In areas where demand for staff outstrips supply, the risk of loosing staff is a motivation for employers to provide job satisfaction.

Human capital theory suggests there is a strong connection between a person's salary and time spent on the job at a company, i.e., the training and experience gained over time increases the likelihood that a person could get a higher paying job elsewhere; it is in a company's interest to increase employee pay over time[155] (one study[1701] of 2,251 IT professionals, in Singapore, found a linear increase in salary over time). Regional employment (see Github–ecosystems/MSA_M2016_CM.R) and pay differences[384] reflect differences in regional cost of living, historical practices (which include gender pay differences[1352]) and peoples' willingness to move.

A study by Couger and Colter[405] investigated approaches to motivating developers working on maintenance activities, the factors included: the motivating potential of the job (based on skill variety required, the degree to which the job requires completion as a whole, the impact of the job on others, i.e., task significance, degree of freedom in scheduling and performing the job, and feedback from the job), and a person's need for personal accomplishment, to be stimulated and challenged.

In some US states, government employee salaries are considered to be public information, e.g., California (see Github–ecosystems/transparentcalifornia.R). A few companies publish employee salaries, so-called *Open salaries*; the information may be available to company employees only, or it may be public, e.g., Buffer.[649]

If the male/female cognitive ability distribution seen in figure 2.5 carries over to software competencies, then those seeking to attract more women into software engineering, and engineering in general, should be targeting the more populous middle competence band, and not the high-fliers. The mass market for those seeking to promote female equality is incompetence; a company cannot be considered to be gender-neutral until incompetent women are equally likely to be offered a job as incompetent men.

3.3.3 Expertise

To become a performance expert, a person needs motivation, time, economic resources, an established body of knowledge to learn from, and teachers to guide; while learning, performance feedback is required.

An established body of knowledge to learn from requires that the problem domain has existed in a stable state for long enough for a proven body of knowledge to become established. The availability of teachers requires that the domain be sufficiently stable that most of what potential teachers have learned is still applicable to students; if the people with the knowledge and skills are to be motivated to teach, they need to be paid enough to make a living.

The practice of software development is a few generations old, and the ecosystems within which developers work have experienced a steady stream of substantial changes; substantial change is written about as-if it is the norm. Anybody who invests in many years of deliberate practice on a specific technology may find there are few customers for the knowledge and skill acquired, i.e., are willing to pay a higher rate to do the job, than that paid to somebody with a lot less expertise. Paying people based on their performance requires a method of reliably measuring performance, or at least differences in performance.

Software developers are not professional programmers any more than they are professional typists; reading and writing source code is one of the skills required to build a software system. Effort also has to be invested in acquiring application domain knowledge and skills, for the markets targeted by the software system.

What level of software development related expertise is worth acquiring in a changing ecosystem? The level of skill required to obtain a job involving software development is judged relative to those who apply for the job, employers may have to make do with whoever demonstrates the basic competence needed. In an expanding market those deemed to have high skill levels may have the luxury of being able chose the work that interests them.

Once a good enough level of programming proficiency is reached, if the application domain changes more slowly than the software environment, learning more about the application domain may provide a greater ROI (for an individual), compared to improving programming proficiency (because the acquired application knowledge/skills have a longer usable lifetime).

People often learn a skill for some purpose (e.g., chess as a social activity, programming because it provides pleasure or is required to get a job done), without aiming to achieve expert performance. Once a certain level of proficiency is achieved, such people stop trying to learn, and concentrate on using what they have learned; in work, and sport, a distinction is made between training for, and performing the activity. During everyday work, the goal is to produce a product, or provide a service. In these situations people need to use well-established methods, to be certain of success, not try new ideas (which potentially lead to failure, or a dead-end). Time spent on non-deliberate practice does not lead to any significant improvement in expertise, although it may increase the fluency of performing a particular subset of skills; computer users have been found to have distinct command usage habits.[1629]

Higher education once served as a signalling system,[1724] used by employers looking to recruit people at the start of their professional careers (i.e., high cognitive firepower was once required to gain a university degree). However, some governments' policy of encouraging a significant percentage of their citizens to study for a higher education degree led to the dilution of university qualifications to being an indication of not below average IQ. By taking an active interest in the employability of graduates with a degree in a STEM related subject,[1652] governments are suffering from cargo-cult syndrome. Knowledge-based businesses want employees who can walk-the-talk, not drones who can hum a few tunes.

3.4 Group dynamics

People may cooperate with others to complete a task that is beyond the capacity of an individual, for the benefit of all those involved. The effectiveness of a group is dependent on cooperation between its members, which makes trust[441] and the enforcement of group norms[190] an essential component of group activity.

In a commercial environment people are paid to work. The economics of personnel selection[1087] are a component of an employer's member selection process; the extent to which employees can select who to work with, will vary.

Groups may offer non-task specific benefits for individuals working within the group, e.g., social status and the opportunity for social learning; discussed in section 3.4.3 and section 3.4.4.

Conflicts of interest can arise between the aims of the group and those of individual members, and those outside the group who pay for its services. Individuals may behave different when working on their own, compared to when working as a member of a group (e.g., social loafing[963]); cultural issues (e.g., individualism vs. collectivism) are also a factor.[517]

When a group of people work together, a shared collection of views, beliefs and rituals (ways of doing things) is evolved,[833] i.e., a culture. Culture improves adaptability. Culture is common in animals, but cultural evolution[1253] is rare; perhaps limited to humans, song birds and chimpanzees.[232]

While overconfidence at the individual level decreases the fitness of the entrepreneur (who does not follow the herd, but tries something new), the presence of entrepreneurs in a group increases group level fitness (by providing information about the performance of new ideas).[183]

People may be active members of multiple groups, with each group involved with different software development projects. Figure 3.17 shows the cumulative number of hours, in weeks since the start of the project, worked by the 47 people involved in the development of an avionics software system. Dashed grey lines are straight lines fitted to three people, and show these people working an average of 3.5, 9 and 13.8 hours per week on this one project.

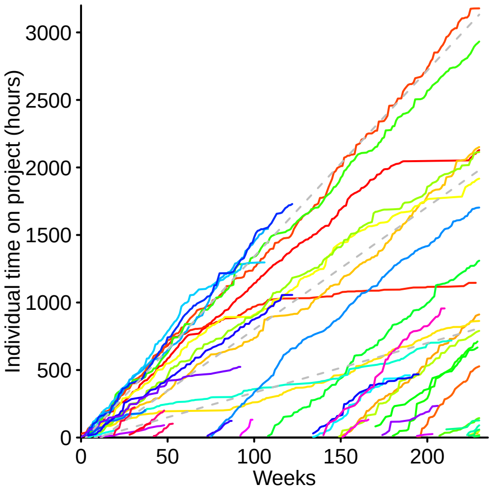


Figure 3.17: The cumulative number of hours worked per week by the 47 individuals involved with one avionics development project; dashed grey lines are straight lines fitted to three individuals. Data from Nichols et al.[1358] Github–Local

3.4.1 Maximizing generated surplus

Given the opportunity, workers will organise their tasks to suit themselves.[1796] The so called *scientific management* approach seeks to find a way of organizing tasks that maximises the surplus value produced by a group (i.e., value produced minus the cost of production),[viii] subject to the constraint that management maintains control of the process. The surplus going to the owners of the means of production.[244]

A study by Taylor[1796] investigated the performance of workers in various industries. He found that workers were capable of producing significantly more than they routinely produced, and documented the problems he encountered in getting them to change their existing practices. Workers have been repeatedly found to set informal quotas amongst themselves,[1931] e.g., setting a maximum on the amount they will produce during a shift.

The scientific management approach was first, and successfully, applied to production where most of the tasks could be reduced to purely manual activities (i.e., requiring little thinking by those who performed them), such as: iron smelting and automobile production. Over the years its use has been extended to all major forms of commercial work, e.g., clerical activities.[1296]

The essence of the scientific management approach is to break down tasks into a small number of component parts, to simplify the component parts so they can be performed by less skilled workers, and then rearrange tasks in a way that gives management control over the production process. Deskilling jobs increases the size of the pool of potential workers, decreasing labor costs and increasing the interchangeability of workers.

Given the almost universal use of this management technique, it is to be expected that managers will attempt to apply it to the production of software,[418, 1030] e.g., chief programmer teams.[118]

The production of software is different from hardware manufacture, in that once the first copy has been created, the cost of reproduction is virtually zero. The human effort invested in creating software systems is primarily cognitive. The division between management and workers is along the lines of what they think about, not between thinking and physical effort.

When the same task is repeatedly performed by different people, it is possible to obtain some measure of average/minimum/maximum individual performance. Task performance improves with practice, and an individual's initial task performance will depend on their prior experience (see section 2.5). Measuring performance based on a single performance of a task provides some indication of minimum performance (see fig 5.27). To obtain information on an individual's maximum performance they have to be measured over multiple performances of the same task (see fig 2.36).

Developer productivity might be quantified in terms of their contribution to the creation and maintenance of a software system. However, until the effort consumed by the many kinds of activities involved in the creation of a software system is quantified, it is not possible to estimate developer productivity.

Performance need not be correlated with productivity, e.g., a developer may deliver a high performance when writing unproductive code.

[viii] Scientific management is not a science of work, it is a science of the management of other people's work.

3.4.2 Motivating members

Developers need motivation to be productive members of a software development team.

Organizations enlist the human need to believe and be part of something larger to motivate and retain members; Apple is perhaps the most well known example.[1479] Quasi-religious narratives are used to enlist developers as participants in a shared mission, e.g., foundation myths and utopian visions. Developer evangelists[967] are employed to seek out third party developers to convert them to believers in a company and its products; providing technical aid opens the door to engagement.

Employee motivation will depend on the relationship they have with their work, which has been categorized as one of: job, career or calling.[1963] Those who treat work as a job see it as providing material benefits (e.g., money), and don't seek other kinds of reward; money as an incentive to motivate employees is a complex issue incentives.[1931] Having a career involves personal investment in the work, with achievements marked with advancement within an organizational structure (advancement may be considered to bring higher social standing, increased authority, and greater self-esteem). People with callings see their work as inseparable from their life, working for the fulfilment that doing the work brings (the work may be seen as socially valuable).

A study by Chandlera and Kapelner[312] investigated the impact of what subjects believed to be meaningfulness of a task on their performance. Workers on Mechanical Turk were paid to complete a task that was framed using one of three descriptions: 1) labelling tumour cells to assist medical researchers (meaningful), 2) the purpose of the task was not given (a control group), and 3) the purpose of the task was not given and subjects were told that their work would be discarded. Subjects in the meaningful group labeled more images and were more accurate; see Github–economics/1210-0962.R.

Some brands acquire a cult following, e.g., Macintosh[159] (and even failed products such as Apple's Newton[1311]). A base of loyal third-party developers who have invested in learning a technology can make it difficult for companies to introduce new, incompatible technologies. The Carbon API was thoroughly entrenched with established with developers creating apps for the Macintosh. Apple's transition from Carbon to a new API (i.e., Cocoa) was significantly helped by the success of the iPhone, whose SDK supported Cocoa (effectively killed continuing interest in Carbon).[855]

Meetings can play a role as rituals of confirmation, reenchantment and celebration.

3.4.3 Social status

The evidence[53] points to a desire for social status as being a fundamental human motive. Higher status (or reputation) provides greater access to desirable things, e.g., interesting projects and jobs.[1293] To be an effective signalling system, social status has to be costly to obtain.[800]

Social animals pay more attention to group members having a higher social status (however that might be measured). People may seek out and pay deference to a highly skilled individual in exchange for learning access. When attention resources are constrained, people may choose to preferentially invest attention on high-status individuals.

Social status and/or recognition is a currency that a group can bestow on members whose activities are thought to benefit the group. The failure of an academic community to treat the production of research related software as worthy of the appropriate academic recognition[1312] may slow the rate of progress in that community; although academic recognition is not always the primary motive for the creation of research software.[853]

A study by Simcoe and Waguespack[1687] investigated the impact of status on the volume of email discussion on proposals submitted to the Internet Engineering Task Force (IETF). A proposal submitted by an IETF working group had a much higher chance of becoming a published RFC, compared to one submitted by a group of individuals (i.e., 43% vs. 7%). The IETF list server distributed proposals with the authors names, except during periods of peak load, when the author list was shortened (by substituting one generic author, "et al"). The, essentially random, shortening of the proposal author list created a natural experiment, i.e., the identity of the authors was available for some proposals, but not others. Proposals unlikely to become a published RFC (i.e., those submitted by individuals), would be expected to receive less attention, unless the list of authors included a high status individual (a working group chairman was assumed to be a high status individual).

An analysis of the number of email posts involving the 3,129 proposals submitted by individuals (mean 3.4 proposals, sd 7.1), found: 1) when a working group chair appeared on the author list, the weighted number of responses increased by 0.8, and 2) when a working group chair name was one of the authors but had been replaced by "et al", the weighted number of responses fell by 1.7; see Github–economics/EtAlData.R.

Figure 3.18 shows the number of proposals receiving a given number of mentions in IETF email discussions, with lines showing fitted regression models; colors denote whether the author list included a working group chairman and whether this information was visible to those receiving the email.

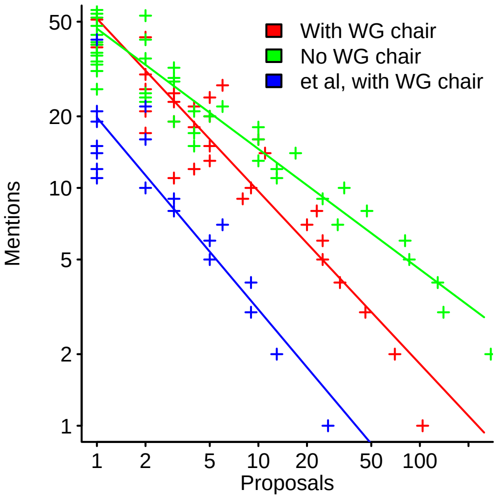


Figure 3.18: Number of proposals receiving a given number of mentions in emails; lines are a fitted regression models of the form: $Mentions \propto Proposals^{-a}$, where a is 0.51, 0.71, and 0.81. Data from Simcoe et al.[1687] Github–Local

3.4.4 Social learning

Learning by observing others, *social learning*, enables animals to avoid the potentially high cost of *individual learning*.[64, 844] A population's average fitness increases when its members are capable of deciding which of two learning strategies, social learning or individual learning, is likely to be the most cost effective[231] (i.e., the cost of individual learning can be invested where its benefit is likely to be maximized). Incremental learning can occur without those involved understanding the processes that underlie the activities they have learned.[478]

Prestige-biased transmission occurs when people select the person with the highest prestige as being the most likely to possess adaptive information[802] for a learner to imitate.

Duplicating the behavior of others is not learning, it is a form of *social transmission*. Following leaders generates a pressure to conform, as does the desire to fit in with a group, known as *conformist transmission*.

In England, milk was once left in open bottles on customer doorsteps; various species of birds were known to drink some of this milk. When bottles were sealed (these days with aluminium foil), some birds learned to peck open milk bottle tops, with the blue tit being the primary scavenger.[598][ix] The evidence suggests those bird species that forage in flocks, have the ability to learn both socially and individually, while species that are territorial (i.e., chase away other members of their species), primarily use individual learning.[1095]

Social learning is a major human skill, where 2.5-year-old children significantly outperform adult chimpanzees and orangutans, our closest primate relatives[809] (performance on other cognitive tasks is broadly comparable).

Just using social learning is only a viable strategy in an environment that is stable between generations of learners. If the environment is likely to change between generations, copying runs the risk of learning skills that are no longer effective. Analysis using a basic model[1223] shows that for a purely social learning strategy to spread in a population of individual learners, the following relation must hold: $u < \frac{c}{b}$, where: u is the probability of environmental change in any generation, b is the benefit of social learning of behavior in the current environment, and c is the cost of learning.

A study by Milgram, Bickman and Berkowitz[1262] investigated the behavior of 1,424 pedestrians walking past a group of people looking up at the 6th-floor window of the building on the opposite side of the street. Figure 3.19 shows the percentage of passers-by lookup up or stopping in response to a crowd of a given size.

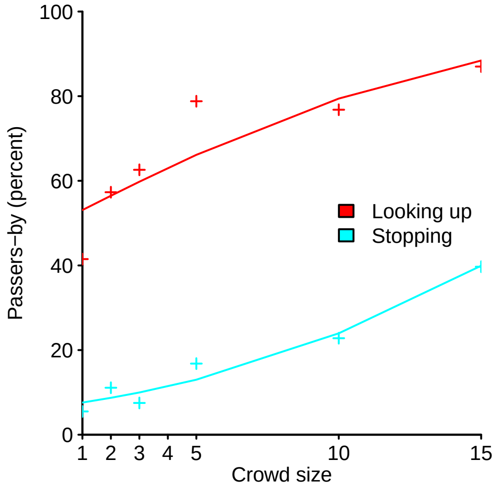


Figure 3.19: Percentage of passers-by looking up or stopping, as a function of group size; lines are fitted linear beta regression models. Data extracted from Milgram et al.[1262] Github–Local

A study by Centola, Becker, Brackbill and Baronchelli[307] investigated the relative size of subgroups acting within a group, needed to change a social convention previously established by group members. Groups containing between 18 and 30 people worked in pairs, with pairs changing after every interaction, performing a naming task; once consistent naming conventions had become established within the group, a subgroup was instructed to use an alternative naming convention. The results found that a committed subgroup containing at least 25% of the group were able to cause the group to switch, to using the subgroup's naming conventions. The rate of change depended on group size and relative size of the subgroup; see Github–economics/Centola-Becker.R.

Discoveries and inventions are often made by individuals, and it might be expected that larger populations will contain more tools and artefacts, and have a more complex cultural repertoire than smaller populations.[1100] The evidence is mixed, with population size having a positive correlation with tool complexity in some cases.[377]

[ix] Your author leaves a plastic beaker for the milkman to place over the bottle, otherwise, within a week the milk bottle top will be regularly opened; something that milkman new to the job have to learn.

Useful new knowledge is not always uniformly diffused out into the world, to be used by others; a constantly changing environment introduces uncertainty[1507] (i.e., noise is added to the knowledge signal), and influential figures may suppress use of the ideas (e.g., some physicists suppressed the use of Feynman diagrams[187]).

Conformist transmission is the hypothesis that individuals possess a propensity to preferentially adopt the cultural traits that are most frequent in the population. Under conformist transmission, the frequency of a trait among the individuals within the population provides information about the trait's adaptiveness. This psychological bias makes individuals more likely to adopt the more common traits than they would under unbiased cultural transmission. Unbiased transmission may be conceptualized in several ways: for example, if an individual copies a randomly selected individual from the population, then the transmission is unbiased; if individuals copy their parents or just their mother, then transmission is unbiased.

The rate of change in the popularity list of baby names, dog breeds and pop music can be fitted[173] by a model where most of the population, N, randomly copy an item on the list, containing L items, and a fraction, μ, select an item not currently on the list. Empirically, the turnover of items on the list has been found to be proportional to: $L\sqrt{\mu}$; a theoretical analysis[548] finds that population size does have some impact, i.e., $L\sqrt{\mu \log \frac{N}{L}}$.

Studies have found that subjects are able to socially learn agreed conventions when communicating within networks having various topologies (containing up to 48 members; the maximum experimental condition).[306] One advantage of agreed conventions is a reduction in communications effort when performing a joint task, e.g., a reduction in the number of words used.[362]

The impact of the transmission of information about new products is discussed in section 3.6.3.

3.4.5 Group learning and forgetting

The performance of groups, like that of individuals (see fig 2.33), improves with practice; groups also forget (in the sense that performance on a previously learned task degrades with time).[894] Industrial studies have focused on learning by doing,[1813] that is the passive learning that occurs when the same, or very similar, product is produced over time.

The impact of organizational learning during the production of multiple, identical units (e.g., a missile or airplane) can be modeled to provide a means of estimating the likely cost and timescale of producing more of the same units.[690] Various models of organizational production[858, 1232] based on connected components, where people are able to find connections between nodes that they can change to improve production, produce the power laws of organizational learning encountered in practice.

There are trade-offs to be made in investment in team learning; there can be improvements in performance and team performance can be compromised.[273]

A study by Nembhard and Osothsilp[1345] investigated the impact of learning and forgetting on the production time of car radios; various combined learning-forgetting models have been proposed, and the quality of fit of these models to the data was compared. Figure 3.20 shows the time taken to build a car radio against cumulative production, with an exponential curve fitted to each period of production (break duration in days appears above the x-axis). Note, build time increases after a break, and both a power law and exponential have been proposed as models of the forgetting process.

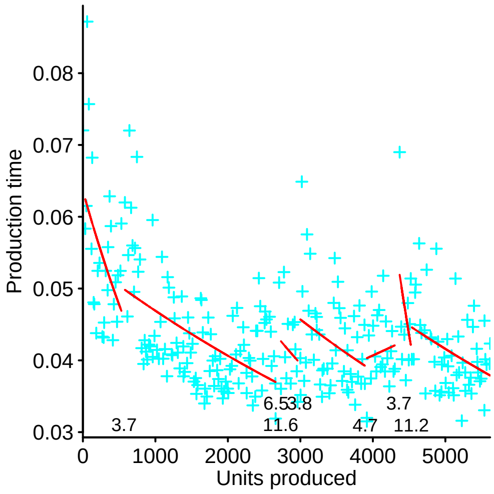


Figure 3.20: Hours required to build a car radio after the production of a given number of radios, with break periods (shown in days above x-axis); lines are regression models fitted to each production period. Data extracted from Nembhard et al.[1345] Github–Local

Writing source code differs from building hardware in that software is easily duplicated, continual reimplementation only occurs in experiments (see fig 2.36). Repetitive activities do occur when writing code, but the repetition is drawn from a wide range of activities sequenced together to create distinct functionality (patterns of code use are discussed in section 7.3).

Figure 3.21 shows the man-hours needed to build three kinds of ships, 188 in total, at the Delta Shipbuilding yard, between January 1942 and September 1945. As experience is gained building Liberty ships the man-hours required, per ship, decreases.[1813] Between May 1943 and February the yard had a contract to build Tankers, followed by a new contract to build Liberty ships, and then Colliers. When work resumes on Liberty ships, the man-hours per ship is higher than at the end of the first contract, presumably some organizational knowledge about how to build this kind of ship had been lost.

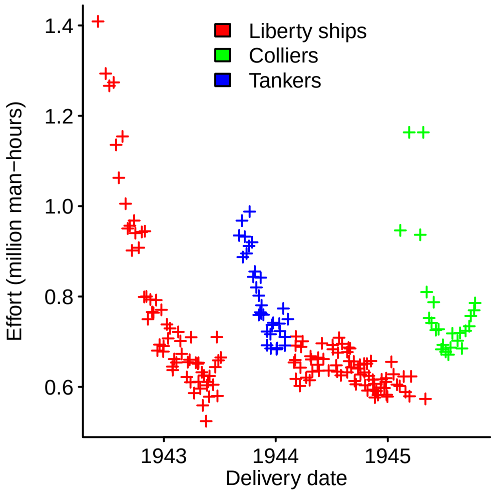


Figure 3.21: Man-hours required to build a particular kind of ship, at the Delta Shipbuilding yard, delivered on a given date (x-axis). Data from Thompson.[1813] Github–Local

When an organization loses staff having applicable experience, there is some loss of performance.[66] It has been proposed[799] that the indigenous peoples' of Tasmania lost valuable skills and technologies because their effective population size shrunk below the level needed to maintain a reliable store of group knowledge (when the sea level rose the islander's were cut off from contact with mainland Australia).

An organization's knowledge and specialist skills are passed on to new employees by existing employees and established practices.

A study by Muthukrishna, Shulman, Vasilescu and Henrich[1320] investigated the transmission of knowledge and skills, between successive generations of teams performing a novel (to team members) task. In one experiment, each of ten generations was a new team of five people (i.e., 50 different people), with every team having to recreate a target image using GIMP. After completing the task, a generation's team members were given 15-minutes to write-up two pages of information, whose purpose was to assist the team members of the next generation. Successive teams in one sequence of ten generations, were given the write-up from one team member of the previous generation, while successive teams in another sequence of ten generations, were given the write-ups from the five members of the previous generation team (i.e., the experiment involved 100 people).

Figure 3.22 shows the rating given to the image produced by each team member, from each of successive generation; lines show fitted regression models. The results suggest that more information was present in five write-ups (blue-green) than one write-up (red), enabling successive generations to improve (or not).

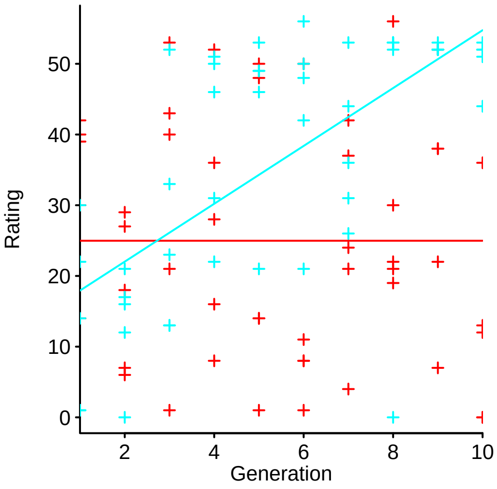


Figure 3.22: Task rating given to members of successive generations of teams; lines are a regression model fitted to the one (red) and five (blue-green) write-up generation sequences. Data from Muthukrishna et al.[1320] Github–Local

Outsourcing of development work, offshore or otherwise, removes the learning-by-doing opportunities afforded by in-house development; in some cases management may learn from the outsource vendor.[309]

3.4.6 Information asymmetry

The term *information asymmetry* describes the situation where one party in a negotiation has access to important, applicable, information that is not available to the other parties. The information is important in the sense that it can be used to make more accurate decisions. In markets where no information about product quality is available, low quality drives out higher quality.[25]

Building a substantial software system involves a huge amount of project specific information; a large investment of cognitive resources is needed to acquire and understand this information. The kinds of specific information involve factors such as: application domain, software development skills and know-how, and existing user work practices.

Vendors bidding to win the contract to implement a new system can claim knowledge on similar systems, but cannot claim any knowledge of a system that does not yet exist. In any subsequent requests to bid to enhance the now-existing system, one vendor can claim intimate familiarity with the workings of the software they implemented. This information asymmetry may deter other vendors from bidding, and places the customer at a disadvantage in any negotiation with the established vendor.

Figure 3.23 shows the ratio of actual to estimated effort for 25 releases of one application, over six years (release 1.1 to 9.1, figures for release 1 are not available).[860] Possible reasons for the estimated effort being so much higher than the actual effort include: cautiousness on the part of the supplier, and willingness to postpone implementation of features planned for the current release to a future release (one of the benefits of trust built up between supplier and customer, in an ongoing relationship, is flexibility of scheduling).

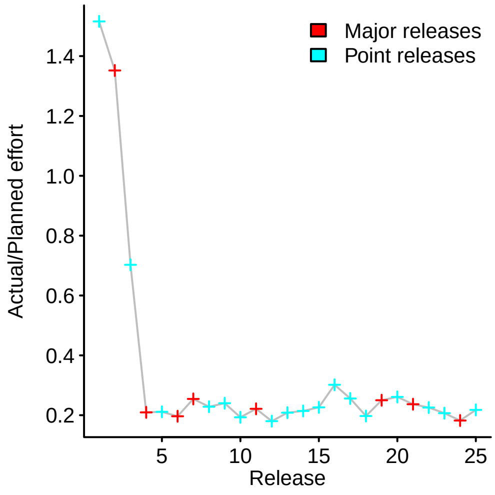


Figure 3.23: Ratio of actual to estimated hours of effort to enhance an existing product, for 25 versions of one application. Data from Huijgens et al.[860] Github–Local

The term *vaporware* is used to describe the practice of announcing software products with a shipment date well in the future.[1508] These announcements are a signalling mechanism underpinned by asymmetrical information, because there may or may not be a real product under development; uses include: keeping existing customer happy that an existing product has a future and deterring potential competitors.[472]

A study by Bayus, Jain and Rao[151] investigated the delay between promised availability, and actual availability, of 123 software products. Figure 3.24 shows that the interval, in months, between the announcement date and promised product availability date has little correlation with the interval between promised and actual delivery date. The fitted regression line shows that announcements promising product delivery in a month or two are slightly more accurate than those promising delivery further in the future.

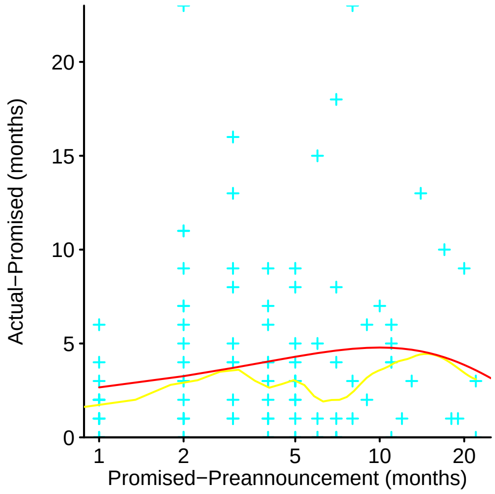


Figure 3.24: Interval between product announcement date and its promised availability date, against interval between promised date and actual date the product became available; lines are a fitted regression model of the form: $A\_P \propto e^{0.3-0.1P\_P+0.8\sqrt{P\_P}}$, and a loess fit. Data from Bayus et al.[151] Github–Local

3.4.7 Moral hazard

A moral hazard occurs when information asymmetry exists, and the more informed party has some control over the unobserved attributes. The traditional example is a manager running a business for those who own the company (the manager has opportunities to enrich himself, at the expense of the owners, who lack the information needed to detect what has happened), but it can also apply to software developers making implementation decisions, e.g., basing their choice on the enjoyment they expect to experience, rather than the likely best technical solution.

Agency theory deals with the conflict of interests of those paying for work to be done, and those being paid to do it.

With so many benefits on offer to the cognitariate, employers need a mechanism for detecting free-riders: employees have to signal hedonic involvement, e.g., by working long hours.[1064]

The reliability of a product may only become visible after extensive use, e.g., the number of faults experienced. Clients may not be able to distinguish the rhetoric and reality of vendor quality management.[1980]

3.4.8 Group survival

Groups face a variety of threats to their survival, including: more productive members wanting to move on (i.e., brain drain), the desire of less productive people to join (i.e., adverse selection), and the tendency of members to shirk (e.g., social loafing).

Groups function through the mutual cooperation between members. Reciprocity: actions towards another involving an expectation of a corresponding cooperative response from the other party is a fundamental component of group cohesion.

Simulations[724] have found that the presence of reciprocity (e.g., x helps/harms y, who in term helps/harms x), and transitivity (e.g., if x and y help each other, and y helps z, then x is likely to help z) in a uniform population, are sufficient for distinct subgroups to form; see Github–economics/2014+Gray+Rand.R.

Cooperation via *direct reciprocity* is only stable when the probability of interacting with a previously encountered member P_i, meets the condition:[1391] $\frac{c}{b} < P_i$, where: c is the cost of cooperation, and b is the benefit received (by one party if they defect, and by both parties if they both cooperate; if an altruist member does not start by cooperating, both parties receive zero benefit); cooperation via *indirect reciprocity*, where decisions are based on reputation (rather than direct experience), is only stable when the probability of knowing a members' reputation, P_k, meets the condition:[1372] $\frac{c}{b} < P_k$.

Members may be tempted to take advantage of the benefits of group membership,[1863] without making appropriate contributions to the group, or failing to follow group norms.[190] If a group is to survive, its members need to be able to handle so-called *free-riders*. Reputation is a means of signalling the likelihood of a person reciprocating a good deed; some combinations of conditions[1392] linking reputation dynamics and reciprocity have been found to lead to stable patterns of group cooperation.

In an encounter between two people, either may choose to cooperate, or not, with the other. A problem known as the

While the analysis of prisoner's dilemma style problems can produce insights, it may not be possible to obtain sufficiently accurate values for the variables used in the model. The following analysis[88] illustrates some of the issues.

In a commercial environment, developers may be vying with each other for promotion, or a pay rise. Developers who deliver projects on schedule are likely to be seen by management as worthy of promotion or a pay rise.

Consider the case of two software developers who are regularly assigned projects to complete, by a management specified date: with probability p, the project schedules are unachievable. If the specified schedule is unachievable, performing Low quality work will enable the project to be delivered on schedule, alternatively the developer can tell management that slipping the schedule will enable High quality work to be performed.

Performing Low quality work is perceived as likely to incur a penalty of Q_1 (because of its possible downstream impact on project completion) if one developer chooses Low, and Q_2

if both developers choose Low. It is assumed that: $Q_1 < Q_2 < C$. A High quality decision incurs a penalty that developers perceive to be C (telling management that they cannot meet the specified schedule makes a developer feel less likely to obtain a promotion/pay-rise).

Let's assume that both developers are given a project, and the corresponding schedule. If either developer faces an unachievable deadline, they have to immediately decide whether to produce High or Low quality work. When making the decision, information about the other developer's decision is not available.

An analysis[88] shows that, both developers choosing High quality is a pure strategy (i.e., always the best) when: $1 - \frac{Q_1}{C} \leq p$, and High-High is Pareto superior to both developers choosing Low quality when: $1 - \frac{Q_2}{C - Q_1 + Q_2} < p < 1 - \frac{Q_1}{C}$.

Obtaining a reasonably estimate for p, C, and Q_1 is likely to be problematic. Inferences drawn from the forms of the equations intended to encourage a High-High decision (e.g., be generous when schedule implementation time, and don't penalise developers when they ask for more time), could have been inferred without using a Prisoner's dilemma model.

Social pressure may not be sufficient to prevent free-riding. Group members have been found to be willing to punish, at a cost to themselves, members who fail to follow group norms (in real-life[1387] and experimental situations[1407]).

A study by Li, Molleman and van Dolder[1117] investigated the impact of prevailing group social norms on the severity with which members punish free-riders. Pairs of subjects played a two-stage game. In the first stage either individual could choose to cooperate (if both cooperate, they each earn 18 points), or defect (i.e., a single defector earns 25 points and the attempted cooperator earns 9; mutual defection earns 16 points). In the second stage, subjects had the opportunity to punish their defecting partner (if the partner had defected) by specifying the number of points to deduct from their partner's earnings; the cost of deduction was a 1-point deduction from the cooperator for every 3-points they wanted to deduct from their defecting partner. Every subject played eleven times, each time with a randomly selected partner.

Prevailing group norms were set by giving subjects information on the behaviour of subjects in an earlier experiment (labeled as the reference group). One group (318 subjects; the CC treatment) read information about the percentage of cooperators in a reference group, the other group (275 subjects; the CP treatment) read information about the number of points deducted by the cooperators in a reference group.

Figure 3.25 shows the mean number of deduction points specified by CC treatment subjects, when told that a given percentage of subjects in a reference group cooperated. Lines show subjects broken out by four response patterns.

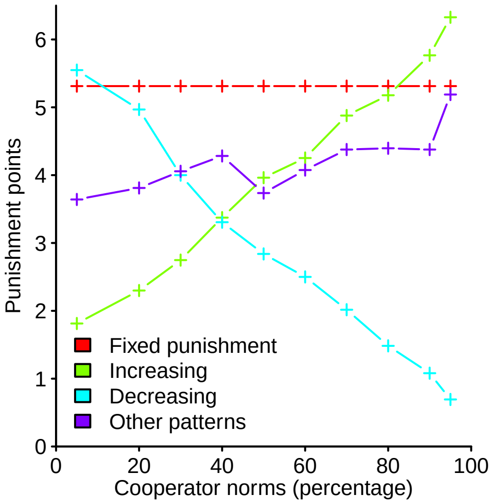


Figure 3.25: Mean number of deduction points specified by subjects told that a given percentage of subjects in a reference group cooperated; broken down by four subject response patterns. Data from Li et al.[1117] Github–Local

For the roughly 60% of subjects who punished their partner in at least one game: in both CC and CP treatments, approximately 30% of subjects always specified the same number of points to deduct. For the CC treatment, just over 33% of subjects deducted more points as the percent of cooperators in the reference group increased, and around 20% deducted fewer points, with 17% following other patterns. For the CP treatment, approximately 45% of subjects deducted fewer points as the mean number of deducted points in the reference group increased, and around 2% deducted fewer points, with 20% following other patterns.

3.4.9 Group problem solving

Group problem solving performance[1081] can be strongly affected by the characteristics of the problem; problem characteristics include:

- unitary/divisible: is there one activity that all members have to perform at the same time (e.g., rope pulling), or can the problem be divided into distinct subcomponents and assigned to individual members to perform?
- maximizing/optimizing: is quantity of output the goal, or quality of output (e.g., a correct solution)?
- interdependency: the method of combining member outputs to produce a group output; for instance: every member completes a task, individual member outputs are added together or averaged, or one member's output is chosen as the group output.

When solving Eureka-type problems (i.e., a particular insight is required), the probability that a group containing k individuals will solve the problem is: $P_g = 1-(1-P_i)^k$, where: P_i is the probability of one individual solving the problem. When the solution involves s subproblems, the probability of group success is (i.e., a combination-of-individuals model): $P_g = \left[1-\left(1-P_i^{1/s}\right)^k\right]^s$. Some studies[1147] have found group performance to have this characteristic.

Studies[1769] have consistently found that for brainstorming problems (i.e., producing as many creative ideas as possible), individuals consistently outperform groups (measured by quantity and quality of ideas generated, per person). Despite overwhelming experimental evidence, the belief that groups outperform individuals has existed for decades.[x]

A study by Bowden and Jung-Beeman[228] investigated the time taken, by individuals, to solve word association problems. Subjects saw three words, and were asked to generate a fourth word that could be combined with each of word to produce a compound word or phrase; for instance, the words SAME/TENNIS/HEAD can all be combined with MATCH.[xi] The 289 subjects were divided into four groups, each with a specified time limit on generating a problem solution (2, 7, 15 and 30 seconds).

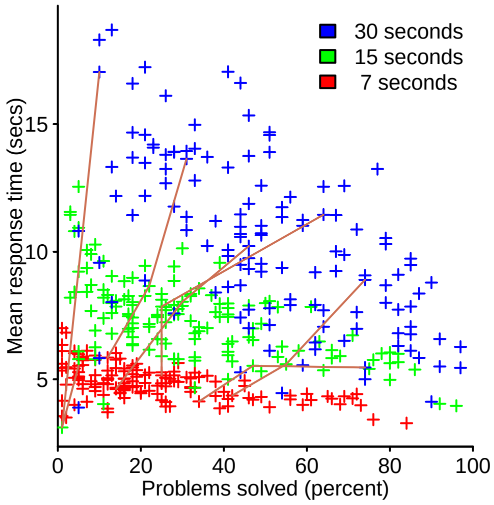


Figure 3.26: Percentage of individuals (x-axis) who correctly generated a solution, against mean response time, for 144 problems; colors denote time limits, and a sample of lines connecting performance pairs for the same program. Data from Bowden et al.[228] Github–Local

Figure 3.26 shows the percentage of subjects who correctly answered a problem against the mean time taken. Each plus corresponds to one of the 144 problems, with colors denoting the time limit (solution time was not collected for the group having a 2-second time limit). Lines link a sample of time/percent performance for answers to the same problem, in each time-limit group.

The results show that the problems have Eureka-type characteristics, with some problems solved quickly by nearly all subjects, and other problems solved more slowly by a smaller percentage of subjects.

The presence of others (e.g., spectators or bystanders) has been found to have a small effect on an individual's performance (i.e., plus/minus 0.5% to 3%).[216]

The information-sampling model[1745] suggests that the likelihood of a group focusing on particular information increases with the number of members who are already aware of it, i.e., information known by one, or a few members, is less likely to be discussed because there are fewer people able to initiate the discussion. The term *hidden profile* has been used to denote the situation where necessary task information is not widely known, and distributed over many group members.

Studies have found that as group size increases, the effort exerted by individuals decreases;[963] the terms *social loafing* and *Ringelmann effect* (after the researcher who first studied[1566] group effort, with subjects pulling on a rope), are used to describe this behavior. Social loafing has been found to occur in tasks that do not involve any interaction between group members, e.g., when asked to generate sound by clapping and cheering in a group, the sound pressure generated by individuals decreases as group size increases.[1079] An extreme form of social loafing is *bystander apathy*, such as when a person is attacked and onlookers do nothing because they assume somebody else will do something.[1078]

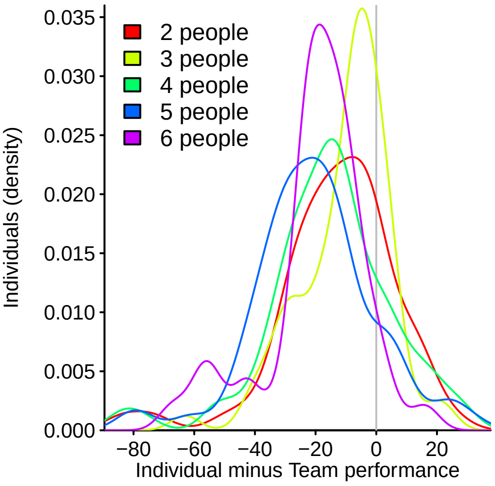


Figure 3.27: Density plot of the difference between mean team mark and individual mark, broken down by team size. Data from Akdemir et al.[24] Github–Local

A study by Akdemir and Ahmad Kirmani[24] analyzed student performance when working in teams and individually, based on marks given for undergraduate projects. Students completed two projects working in a team and one project working alone, with all team members given the same mark for a project. Figure 3.27 shows a density plot of the difference between mean team mark and individual mark, for the 386 subjects, broken down by team size.

A study by Lewis[1109] investigated the number of ideas generated by people working in groups or individually. Subjects were given a task description and asked to propose as many methods as possible for handling the various components of the task. Experiment 1 involved groups of 1, 2, 4 and 6 people, who were given 50 minutes to produce ideas (every 5-minutes groups recorded the total number of ideas created).

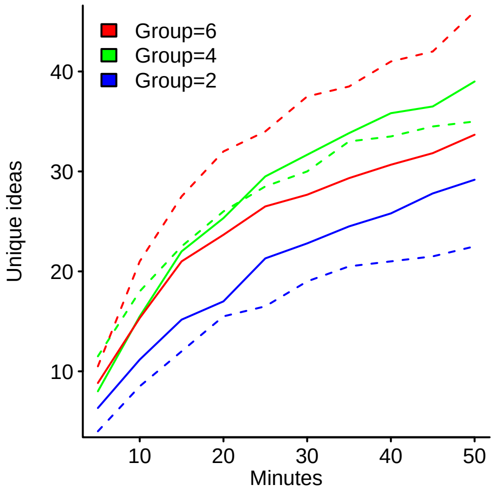


Figure 3.28: Average number of ideas produced by groups of a given size, at 5-minute interval elapsed time; dashed lines are nominal groups created by aggregating individual ideas. Data from Lewis.[1109] Github–Local

Figure 3.28 shows the average number of ideas produced by each group size. The solid lines show actual groups, the dashed lines show nominal groups, e.g., treating two subjects working individually as a group of two and averaging all their unique ideas.

[x] This belief in the benefits of brainstorming groups has been traced back to a book published by an advertising executive in the 1950s.

[xi] Synonymy (same = match), compound word (matchhead), and semantic association (tennis match).

3.4.10 Cooperative competition

Within ecosystems driven by strong network effects, there can be substantial benefits in cooperating with competitors, e.g., the cost of not being on the winning side of the war, to set a new product standard, can be very high,[1657] and sharing in a winner take-all market is an acceptable outcome.

Software systems need to be able to communicate with each other, agreed communication protocols are required. The communication protocols may be decided by the dominant vendor (e.g., Microsoft with its Server protocol specifications[1260]), by the evolution of basic communication between a few systems to something more complicated involving many systems, or by interested parties meeting together to produce an agreed specification. The components of hardware devices also need to interoperate, and several hundred interoperability standards are estimated to be involved in a laptop.[191]

Various standards' bodies, organizations, consortia, and groups have been formed to document agreed-upon specifications. Committee members are often required to notify other members of any patents they have, that impinge on the specification being agreed to. Organizations that failed to reveal patents they hold, if they subsequently attempt to extract royalties from companies implementing the agreed specification, may receive large fines and licensing their patents under reasonable terms may be government enforced.[576]

A study by Simcoe[1685] investigated the production of communication specifications (i.e., RFCs) by the Internet Engineering Task Force (IETF) between 1993 and 2003 (the formative years of the Internet). Figure 3.29 shows that the time taken to produce an RFC having the status of a Standard, increased as the percentage of commercial membership of the respective committee increased, but there was no such increase for RFCs not having the status of a standard.

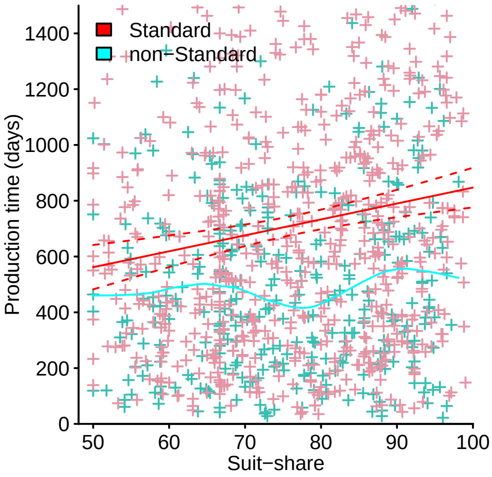


Figure 3.29: Time taken to publish an RFC having Standard or non-Standard status, for IETF committees having a given percentage of commercial membership (i.e., people wearing suits); lines are a fitted regression model with 95% confidence intervals (red), and a loess fit (blue/-green). Data from Simcoe.[1685] Github–Local

If several companies have to build a large software system that they don't intend to sell as a product (i.e., it's for in-house use), a group of companies may agree to cooperate in building the required system. The projects to build these systems involve teams containing developers from multiple companies.

A study by Teixeira, Robles and González-Barahona[1800] investigated the evolution of company employees working on the releases of the OpenStack project, between 2010 and 2014. Figure 3.30 shows the involvement of top ten companies, out of over 200 organizations involved, as a percentage of employed developers working on OpenStack.

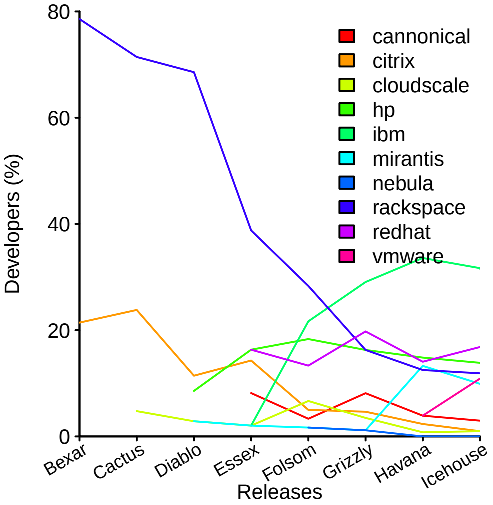


Figure 3.30: Percentage of developers, employed by given companies, working on OpenStack at the time of a release (x-axis). Data from Teixeira et al.[1800] Github–Local

3.4.11 Software reuse

Reuse of existing code has the potential to save time and money, and reused code may contain fewer mistakes than newly written code (i.e., if the reused code has been executed, there has been an opportunity for faults to be experienced and mistakes fixed). Many equations detailing the costs and benefits of software reuse have been published,[1263] and what they all have in common is not being validated against any evidence.

Reuse of preexisting material is not limited to software, e.g., preferential trade agreements.[31]

Multiple copies of lexically, semantically, or functionally similar source code may be referred to as *reused code*, *duplicate code* or as a *clone*.[xii] Investigating whether cloned code is more fault prone than non-cloned code[845] is something of a cottage industry, but these studies often fail to fully control for mistakes in non-cloned versions of the code. There are specific mistake patterns that are the result of copy-and-paste errors.[165]

Creating reusable software can require more investment than that needed to create a non-reusable version of the software. Without an expectation of benefiting from the extra investment, there is no incentive to make it. In a large organization, reuse may be worthwhile at the corporate level, however the costs and benefits may be dispersed over many groups who have no incentive for investing their resources for the greater good.[591]

Reasons for not reusing code include: the cost of performing due diligence to ensure that intellectual property rights are respected (clones of code appearing on Stack Overflow[xiii] have been found in Android Apps[51] having incompatible licenses, and Github

[xii] Clone is a term commonly used in academic papers.

[xiii] Example code on Stack Overflow is governed by a Creative Commons Attribute-ShareAlike 3.0 Unported license.

projects[123]), ego (e.g., being recognized as the author of the functionality), and hedonism (enjoyment from inventing a personal wheel, creates an incentive to argue against using somebody else's code). Reusing significant amounts of code complicates cost and schedule estimation;[385] see fig 5.31.

Reuse first requires locating code capable of being cost effectively reused to implement a given requirement. Developers are likely to be familiar with their own code, and the code they regularly encounter. The economics of reusable software are more likely to be cost effective at a personal consumption level, and members of a group may feel obligated to make an investment in the group well-being.

A study by Li, Lu, Myagmar and Zhou[1119] investigated copy-and-pasted source within and between the subsystems of Linux and FreeBSD. Table 3.2 shows that Linux subsystems contain a significant percentage of replicated sequences of their own source; replication between subsystems is less common (the same pattern was seen in FreeBSD 5.2.1).

subsystem	**arch**	**fs**	**kernel**	**mm**	**net**	**sound**	**drivers**	**crypto**	**others**	**LOC**
arch	*25.1*	1.4	0.5	0.3	1.1	1.3	3.2	0.1	0.8	724,858
fs	1.4	*16.5*	0.6	0.5	1.7	1.2	2.2	0.0	0.7	475,946
kernel	3.0	1.8	*7.9*	0.6	2.3	1.6	2.8	0.1	0.8	30,629
mm	2.6	2.2	0.8	*6.2*	1.7	1.1	2.0	0.0	0.7	23,490
net	1.8	2.5	1.1	0.7	*20.7*	2.1	3.7	0.1	1.0	334,325
sound	2.3	2.0	1.0	0.6	2.2	*27.4*	4.6	0.2	1.1	373,109
drivers	2.3	1.7	0.6	0.4	1.8	2.0	*21.4*	0.1	0.6	2,344,594
crypto	2.3	2.2	0.3	0.1	1.1	1.5	2.5	*26.1*	2.2	9,157
others	3.8	1.9	0.8	0.4	1.7	1.5	2.6	0.3	*15.2*	49,016

Table 3.2: Percentage of a subsystem's source code cloned within and across subsystems of Linux 2.6.6. Data from Li et al.[1119]

A study by Wang[1902] investigated the survival of clones (a duplicate sequence of 50 or more tokens; Type 1 clones are identical, ignoring whitespace and comments, while Type 2 clones allow identifiers and literal values to be different, and Type 3 clones allow non-matching gaps) in the Linux high/medium/low level SCSI subsystems (the architecture of this system has three levels). Figure 3.31 shows the clone survival curves, which all have an initial half-life of around 18 months, followed by much longer survival lifetimes.

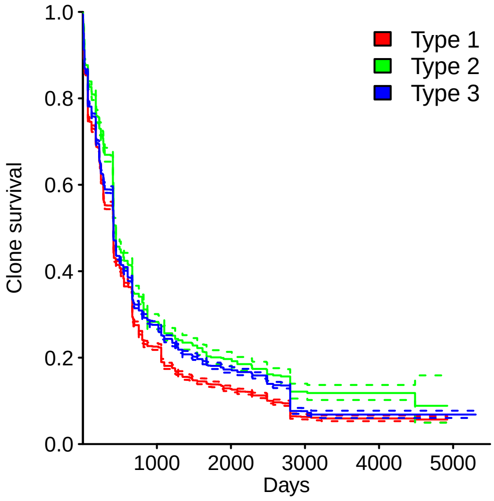


Figure 3.31: Survival curves of type I, II and III clones in the Linux high/medium/low level SCSI subsystems; dashed lines are 95% confidence intervals. Data from Wang.[1902] Github–Local

3.5 Company economics

The value of a company[245] is derived from two sources: tangible goods such as buildings it owns, equipment and working capital, and intangible assets which are a product of knowledge (e.g., employee know-how, intellectual property[20] and customer switching costs); see fig 1.8.

Governments have been relatively slow to include intangible investments in the calculation of GDP[1103] (starting in 1999 in the US and 2001 in the UK), and different approaches to valuing software[19] has led to increased uncertainty when comparing country GDP (McGee[1225] discusses accounting practices, for software, in the UK and US during the 1970s early 1980s). The accounting treatment of intangibles depends on whether it is purchased from outside the company, requiring it to be treated as an asset, or generated internally where is often treated as an expense.[1306] A company's accounts may only make sense when its intangible assets are included in the analysis.[863]

The financial structure of even relatively small multinational companies, is likely to be complicated.[1029]

3.5.1 Cost accounting

The purpose of cost accounting is to help management make effective cost control decisions. In traditional industries the primary inputs are the cost of raw materials (e.g., the money needed to buy the materials needed to build a widget), and labor costs (e.g., the money paid to the people involved in converting the raw materials into a widget); other costs might include renting space where people can work, and the cost of consumables such as electricity.

The production of software systems is people driven, and people are the primary cost source.

The total cost of a software developer can be calculated from the money they are paid, plus any taxes levied by the government, on an employer, for employing somebody (e.g., national insurance contributions in the UK[xiv]); the term *fully loaded cost* is sometimes used.

3.5.2 The shape of money

Money is divided up, and categorized, in various ways for a variety of different reasons. Figure 3.32 illustrates the way in which UK and US tax authorities require registered companies to apportion their income to various cost centers.

Within a company, the most senior executives allocate a budget for each of the organizational groups within the company. These groups, in turn, divide up their available budget between their own organizational groups, and so on. This discrete allocation of funds can have an impact on how software development costs are calculated.

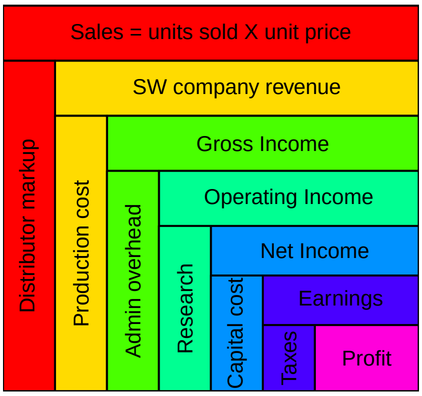


Figure 3.32: Accounting practice for breaking down income from sales, and costs associated with major business activities. Github–Local

Take the example of a development project, where testing is broken down into two phases, i.e., integrating testing and acceptance testing, each having its own budget, say B_i and B_a.

One technique sometimes used for measuring the cost of faults is to divide the cost of finding them (i.e., the allocated budget), by the number of faults found. For instance, if 100 unique fault experiences occur during integration testing, the cost per fault in this phase is $\frac{B_i}{100}$, and if five unique fault experiences occur during acceptance testing, the cost per fault in this phase is $\frac{B_a}{5}$.

One way of reducing the cost per fault experienced during acceptance testing would be to reduce the effectiveness of integration testing. Because, for a fixed budget, the cost per fault decreases as the number of faults experienced increases.

This accountancy-based approach to measuring the cost of faults, creates the impression that it is more costly to find faults later in the process, compared to finding them earlier.[222] This conclusion is created through the artefact of a fixed budget, along with the typical case that fewer unique fault experiences occur later in the development process.

Time taken to fix faults may less susceptible to accounting artefacts, provided the actual time is used (i.e., not the total allocated time). Figure 3.33 shows the average number of days used to fix a reported fault in a given phase (x-axis), caused by a mistake that had been introduced in an earlier phase (colored lines), based on 38,120 faults in projects at Hughs Aircraft;[1946] also see fig 6.43.

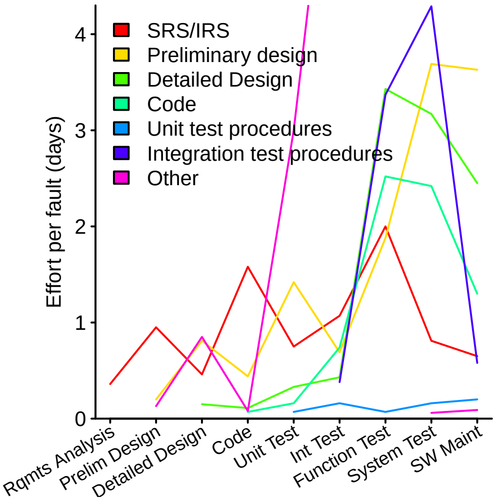


Figure 3.33: Average effort (in days) used to fix a fault experienced in a given phase (x-axis) caused by a mistake that had been introduced in an earlier phrase (colored lines), introduced in an earlier phase (total of 38,120 defects in projects at Hughes Aircraft). Data extracted from Willis et al.[1946] Github–Local

3.5.3 Valuing software

Like any other item, if no one is willing to pay money for the software, it has zero sales value. The opportunity cost of writing software from scratch, along with the uncertainty of being able to complete the task, and the undocumented business rules it implements, is what makes it possible for existing software to be worth significantly more than the original cost of creating it (along with any installed based). However, for accounting purposes, software may be valued in terms of the cost of producing it.

Various approaches to valuing software, and any associated intellectual property are available.[1934]

An organization seeking to acquire a software system has the option of paying for its implementation, and the cost of creating a software system is one approach to valuing it. However, this approach to valuing an existing system assumes that others seeking to reimplement it have access to the application domain expertise needed to do the job (along with successfully hiring developers with the necessary skills). A reimplementation has a risk premium attached, along with a lead time (which may be the crucial factor in a rapidly changing market).

A study by Gayek, Long, Bell, Hsu and Larson[653] obtained general effort/size information on 452 military/space projects. Figure 3.34 shows executable statements and the developer effort (in months) used to create them (the range of developer salaries is known).

[xiv] This was zero-rated up to a threshold, then 12% of employee earnings, increasing on reaching an upper threshold; at the time of writing.

Lines are quantile regression fits at 10 and 90% for the one of the application domains, and show a factor of five variation in developer effort (i.e., costs) for the same ESLOC.

Commercial companies are required to keep accurate financial accounts, whose purpose is to provide essential information for those with a financial interest in the company, including governments seeking to tax profits. Official accounting organizations have created extensive, and ever-changing, rules for producing company accounts, including methods for valuing software and other products of intellectual effort.[794] In the US, the Financial Accounting Standards Board has issued an accounting standard "FASB Codification 985-20" (FAS 80[594] was used until 2009) covering the "Accounting for the Costs of Computer Software to Be Sold, Leased, or Otherwise Marketed". This allows software development costs to be treated either as an expense or capitalized (i.e., treated like the purchase of an item of hardware). An expense is tax-deductible in the financial year in which it occurs, but the software does not appear in the company accounts as having any value; the value of a capitalized item is written down over time (i.e., a percentage of the value is tax-deductible over a period of years), but has a value in the company accounts.[989]

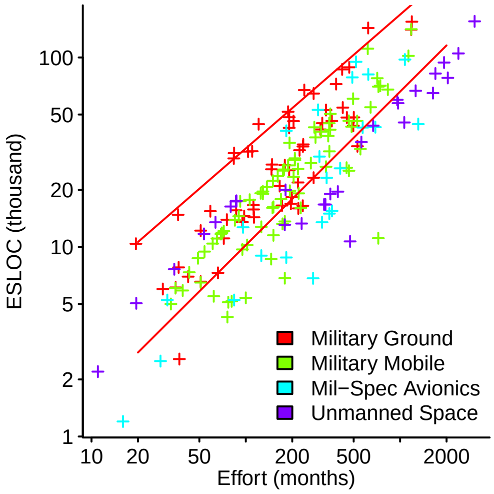


Figure 3.34: Months of developer effort needed to produce systems containing a given number of lines of code, for various application domains; lines are quantile regression fits at 10 and 90%, for one application domain. Data from Gayek et al.[653] Github–Local

The decision on how software development costs appear in the company accounts can be driven by the desire to project a certain image to interested outsiders (e.g., the company is worth a lot because it owns valuable assets[4]), or to minimise tax liabilities. A study by Mulford and Roberts[1307] of 207 companies (primarily industry classification SIC 7372) in 2006, found that 30% capitalized some portion of their software development, while a study by Mulford and Misra[1306] of 100 companies in 2015, found 18% capitalizing their development.

Software made available under an Open Source license may be available at no cost, but value can be expressed in terms of replacement cost, or the cost of alternatives.

For accounting purposes, the cost of hardware is depreciated over a number of years. While software does not wear out, it could be treated as having a useful lifespan (see section 4.2.2).

3.6 Maximizing ROI

Paying people to write software is an investment, and investors want to maximise the return on their investments.

To be viable, a commercial product requires a large enough market capable of paying what it takes. One study[980] of Android Apps found that 80% of reviews were made by people owning a subset of the available devices (approximately 33%). Given the cost of testing an App in the diverse Android ecosystem, the ROI can be increased by ignoring those devices that are owned by a small percentage of the customer base.

Some people write software to acquire non-monetary income, but unless this is the only income sought (e.g., income derived purely from the creation process, or the pride in releasing an App), maximising returns will be of interest.

A study by Li, Bissyandé and Klein[1113] investigated the release history of 3,271,646 Apps in Google Play (the App market does not make available a release history, and the data collection process may have missed some releases). Figure 3.35 shows the number of Apps in Google Play having a given number of releases, along with a regression line fitted to the first 20 releases.

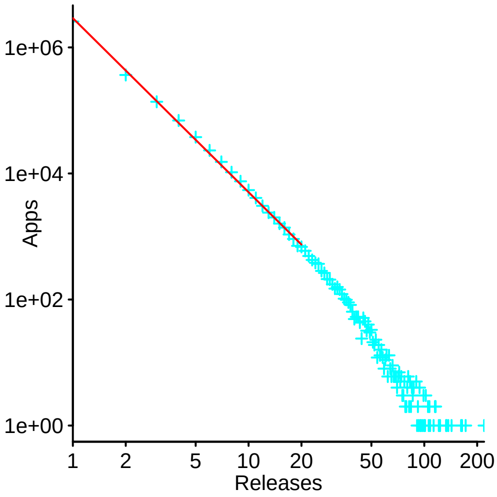


Figure 3.35: Number of Apps in the Google playstore having a given number of releases; line is a fitted regression model of the form: $Apps \propto Releases^{-2.8}$. Data kindly provided by Li.[1113] Github–Local

Accounting issues around choices such as purchase vs. leasing, contractors vs. employees, and making efficient use of income held offshore,[192] are outside the scope of this book.

3.6.1 Value creation

A business model specifies an organization's value creation strategy.[1622]

Many businesses structure their value creation processes around the concept of a value chain; in some software businesses value creation is structured around the concept of a value network.[1735]

A *value chain* is the collection of activities that transform inputs into products. "A firm's value chain and the way it performs individual activities are a reflection of its history, its strategy, its approach to implementing its strategy, and the underlying economics of the activities themselves.”.[1488] Governments tend to view[453] creative value chains as applying to artistic output, and in the digital age artistic output shares many of the characteristics of software development.

A value chain for bespoke software development might be derived from the development methodology used to produce a software system, e.g., the phases of the waterfall model.[213]

The business model for companies selling software products would include a value chain that included marketing,[326] customer support and product maintenance.

A company whose business is solving customer problems (e.g., professional services) might be called a *value shop*.[1735]

In a two-sided market, value flows from one side to the other. Companies create value by facilitating a network relationship between their customers, e.g., Microsoft encourage companies to create applications running under Microsoft Windows, and make money because customers who want to run the application need a copy of Windows; applications and platforms are discussed in section 4.5.

3.6.2 Product/service pricing

Vendors can set whatever price they like for their product or service. The ideal is to set a price that maximises profit, but there is no guarantee that income from sales will cover the investment made in development and marketing. Even given extensive knowledge of potential customers and competitors, setting prices is very difficult.[1330, 1680]

Like everything else, software products are worth what customers are willing to pay. When prices quoted by vendors are calculated on an individual customer basis, inputs considered include how much the customer might be able to pay.[1625] Shareware, software that is made freely available in the hope that users will decide it is worth paying for,[1787] is one of the purest forms of customer payment decision-making.

Customer expectation, based on related products, acts as a ballpark from which to start the estimation process; figure 3.36 shows the relative price/performance used by Intel for one range of processors. It may be possible to charge more for a product that provides more benefits, or perhaps the price has to match that of the established market leader, and the additional benefits are used to convince existing customers to switch.

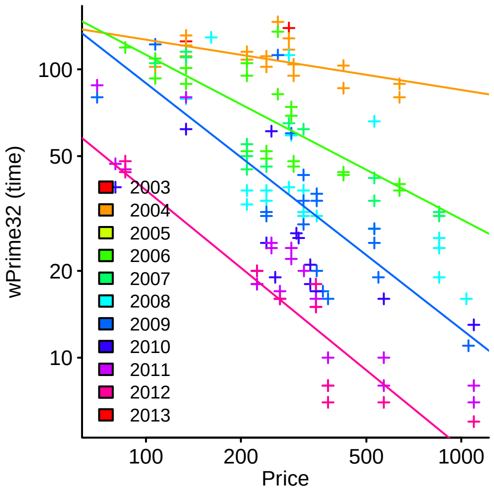


Figure 3.36: Introductory price and performance (measured using wPrime32 benchmark; lower is better) of various Intel processors between 2003-2013. Data from Sun.[1773] Github–Local

Figure 3.37 shows the prices charged by several established C/C++ compiler vendors. In 1986[xv] Zorland entered the market with a £29.95 C compiler, an order of magnitude less than what most other vendors were charging at the time. This price was low enough for many developers to purchase a copy out of company petty cash, and Zorland C quickly gained a large customer base. Once the product established a quality reputation, Zorland were able to increase prices to the same level as those charged by other major vendors (Zorland sounds very similar to the name of another major software vendor of the period, Borland. Letters from company lawyers are hard evidence that a major competitor thinks your impact on their market is worthy of their attention; Zorland became Zortech).[xvi]

A study by Viard[1878] investigated software retail and upgrade pricing, in particular C and C++ compilers available under Microsoft DOS and Windows, between 1987 and 1998. Figure 3.37 shows that the retail price of C/C++ compilers, running under Microsoft DOS and Windows, had two bands that remained relatively constant. Microsoft had a policy of encouraging developers to create software for Windows,[1474] on the basis that sales of applications would encourage sales of Windows, and significantly greater profits would be made from the increased volume of Windows’ sales, compared to sales of compilers. Microsoft’s developer friendly policy kept the price of C/C++ compilers under Windows down, compared to other platforms.[xvii]

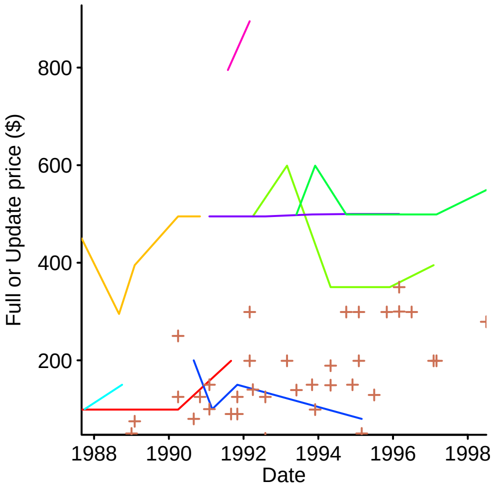


Figure 3.37: Vendor C and C++ compiler retail price (different line for each product), and upgrade prices (pluses) for products available under MS-DOS and Microsoft Windows between 1987 and 1998. Data kindly provided by Viard.[1878] Github–Local

Many companies give managers the authority to purchase low cost items, with the numeric value of low price increasing with seniority. The upper bound on the maximum price that can be charged for a product or service, that can be sold relatively quickly to businesses,

[xv] Richard Stallmann’s email announcing the availability of the first public release of gcc was sent on 22 March 1987.

[xvi] Being in the compiler business your author had copies of all the major compilers, and Zorland C was the compiler of choice for a year or two. Other low price C compiler vendors were unable to increase their prices because of quality issues relative to other products on the market, e.g., Mix C.

[xvii] The 1990s was the decade in which gcc, the only major open source C/C++ compiler at the time, started to be widely used.

is the purchasing authority of the target decision maker. Once the cost of an item reaches some internally specified value (perhaps a few thousand pounds or dollars), companies require that a more bureaucratic purchase process be followed, perhaps involving a purchasing committee, or even the company board. Navigating a company's bureaucratic processes requires a sales rep, and a relatively large investment of time, increasing the cost of sales, and requiring a significant increase in selling price; a price dead-zone exists between the maximum amount managers can independently sign-off, and the minimum amount it is worth selling via sales reps.

Supply and demand is a commonly cited economic pricing model. The supply curve is the quantity of an item that a supplier is willing, and able, to supply (i.e., sell) to customers at a given price, and the demand curve is the quantity of a product that will be in demand (i.e., bought by customers) at a given price. If these two curves intersect, the intersection point gives the price/quantity at which suppliers and customers are willing to do business; see figure 3.38.

Events can occur that cause either curve to shift. For instance, the cost of manufacturing an item may increase/decrease, shifting the supply curve up/down on the price axis (for software, the cost of manufacturing is the cost of creating the software system); or customers may find a cheaper alternative, shifting the demand curve down on the price axis.

In some established markets, sufficient historic information has accumulated for reasonably accurate supply/demand curves to be drawn. Predicting the impact of changing circumstances on supply-demand curves remains a black art for many products and services. Software is a relatively new market, and one that continues to change relatively quickly. This changeability makes estimating supply/demand curves for software products little more than wishful thinking.

Listed prices are often slightly below a round value, e.g., £3.99 or £3.95 rather than £4.00. People have been found to perceive this kind of price difference to be greater than the actual numerical difference[1810] (the value of the left digit, and the numeric distance effect have been used to explain this behavior). Figure 3.39 shows the impact that visually distinct, small price differences, have on the rate of sale of items listed on Gumroad (a direct to consumer sales website). Other consumer price perception effects include precise prices vs. round prices.[1811]

Other pricing issues include site licensing, discounting and selling through third-parties.[1663] The lack of data prevents these issues being discussed here.

The price of a basket of common products, over time, is used to calculate the consumer price index, and changes in product prices over time can be used to check the accuracy of official figures.[1426] Products may evolve over time, with new features being added, and existing features updated; techniques for calculating quality adjusted prices have been developed;[1773] see figure 3.36.

3.6.3 Predicting sales volume

The likely volume of sales is a critical question for any software system intended to be sold to multiple customers.

When one product substitutes another, new for old, market share of the new product is well fitted by a logistic equation,[599] whose maximum is the size of the existing market. Software may be dependent on the functionality provided by the underlying hardware, which may be rapidly evolving.[1004] Figure 3.40 shows how software sales volume lags behind sales of the hardware needed to run it. An installed hardware base can be an attractive market for software.[1190]

Estimating the likely sales volume for a product intended to fill a previously unmet customer need, or one that is not a pure substitution, is extremely difficult (if not impossible).[1766]

The Bass model[139,1441] has been found to fit data on customer adoption of a new product and successive releases, and has been used to make short term sales predictions[1964] (the following analysis deals with the case where customers are likely to make a single purchase; the model can be extended to handle repeat sales[140]). The model divides customers into innovators, people who are willing to try something new, and imitators, people who buy products they have seen others using (this is a diffusion model). The interaction

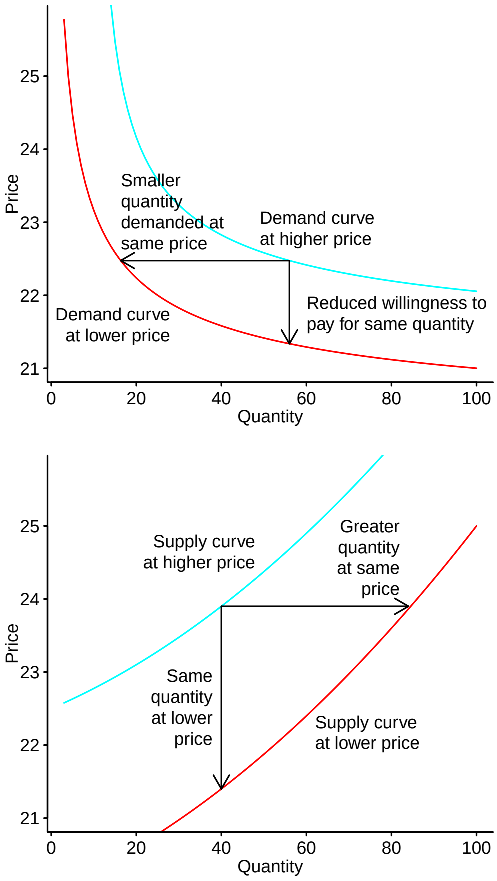


Figure 3.38: Example supply (lower) and demand (upper) curves. Github–Local

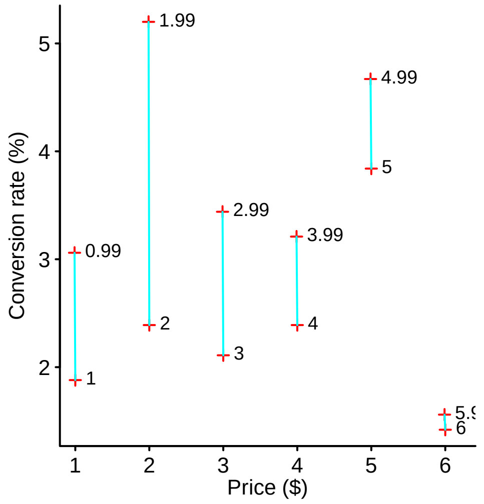


Figure 3.39: Rates at which product sales are made on Gumroad at various prices; lines join prices that differ in 1¢, e.g., $1.99 and $2. Data from Nichols.[1356] Github–Local

between innovators, imitators and product adoption, at time t, is given by the following relationship:

$\frac{f(t)}{1-F(t)} = p + qF(t)$, where: $F(t)$ is the fraction of those who will eventually adopt (i.e., have purchased) by time t, $f(t)$ is the probability of purchase at time t (i.e., the derivative of $F(t)$, $f(t) = dF(t)/dt$), p the probability of a purchase by an innovator, and q the probability of a purchase by an imitator.

This non-linear differential equation[xviii] has the following exact solution, for the cumulative number of adopters (up to time t), and the instantaneous number of adopters (at time t):

$$F(t) = \frac{1-e^{-(p+q)t}}{1+\frac{q}{p}e^{-(p+q)t}} \qquad \text{and} \qquad f(t) = \frac{(p+q)^2}{p}\frac{e^{-(p+q)t}}{\left[1+\frac{q}{p}e^{-(p+q)t}\right]^2}$$

Actual sales, up to, or at, time t, are calculated by multiplying by m, the total number of product adopters.

When innovators dominate (i.e., $q \leq p$), sales decline from an initial high-point; when imitators dominate (i.e., $q > p$), peak sales of $m\left(\frac{1}{2} - \frac{p}{2q}\right)$ occurs at time $\frac{m}{p+q}\log\frac{q}{p}$, before declining; the expected time to adoption is: $E(T) = \frac{m}{q}\log\frac{p+q}{p}$.

The exact solution applies to a continuous equation, but in practice sales data is discrete (e.g., monthly, quarterly, yearly). In the original formulation the model was reworked in terms of a discrete equation, and solved using linear regression (to obtain estimates for p, q); however, this approach produces biased results. Benchmarking various techniques[1441] finds that fitting the above non-linear equation to the discrete data produces the least biased results.

Vendors want reliable estimates of likely sales volume as early in the sales process as possible, but building accurate models requires data covering a non-trivial range of the explanatory variable (time in this case). Figure 3.41 shows the number of Github users during its first 58 months, and Bass models fitted to the first 24, 36, 48 and 58 months of data.[xix].

The Bass model includes just two out of the many possible variables that could affect sales volume, and models that include more variables have been developed.[1273] Intel have used an extended Bass Model to improve forecasting of design wins.[1964]

The Bass model can be extended to handle successive, overlapping generations of a product; the following example is for two generations:[1370]

$$S_1(t) = F_1(t)m_1 - F_2(t-\tau_2)F_1(t)m_1 = F_1(t)m_1(1-F_2(t-\tau_2))$$
$$S_2(t) = F_2(t-\tau_2)(m_2 + F_1(t)m_1)$$

where: $S_i(t)$ are all sales up to time t for product generation i, m_i the total number who adopt generation i, and τ_2 the time when the second generation can be bought; p_i and q_i are the corresponding purchase probabilities for each generation.

The Bass model uses two of the factors driving product sales, other factors that can play a significant role include advertising spend, and variability in market size caused by price changes. Monte Carlo simulation can be used to model the interaction of these various factors.[1856]

How much value, as perceived by the customer, does each major component add to a system? A technique for obtaining one answer to this question, is Hedonic regression (this approach is used to calculate the consumer price index): this fits a regression model to data on product price and product configuration data. A study by Stengos and Zacharias[1750] performed a hedonic analysis of the Personal Computer market, based on data such as price, date, CPU frequency, hard disc size, amount of RAM, screen width and presence of a CD; see Github–economics/0211_Computers.R.

The ease with which software can be copied makes piracy an important commercial issue. Studies[676, 1912] have extended the Bass model to include the influence of users of pirated software, on other people's purchasing decisions (e.g., Word processors and Spreadsheet programs between 1987 and 1992). The results, based on the assumptions made

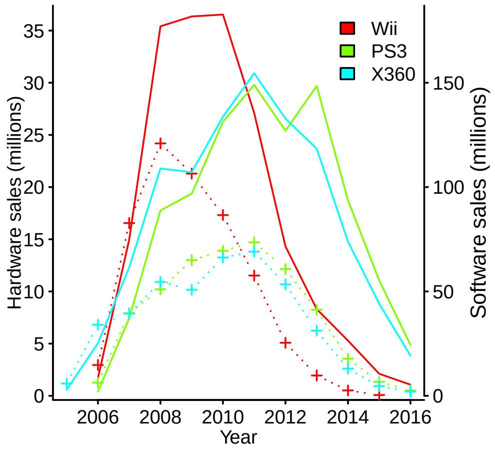


Figure 3.40: Sales of game software (solid lines) for the corresponding three major seventh generation hardware consoles (dotted lines). Data from VGChartz.[1877] Github–Local

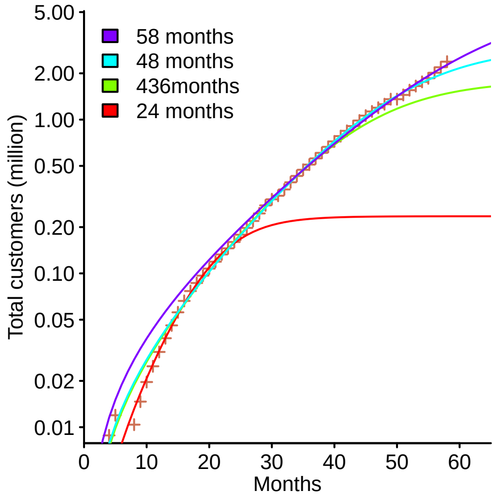


Figure 3.41: Growth of Github users during its first 58 months, with Bass models fitted to data up to a given number of months. Data from Irving.[886] Github–Local

[xviii] It has the form of a Riccati equation.

[xix] Chapter 11 provides further evidence that predictions outside the range of data used to fit a model can be very unreliable.

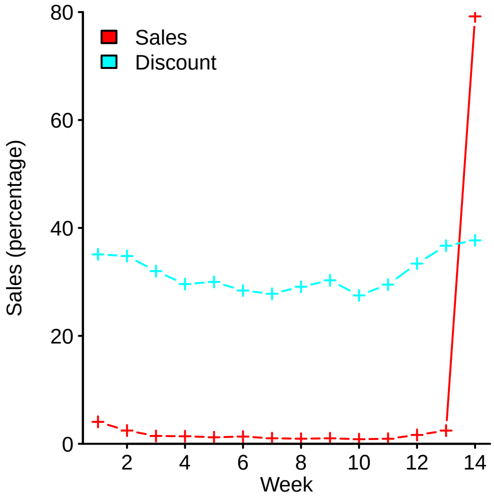


Figure 3.42: Percentage of sales closed in a given week of a quarter, with average discount given. Data from Larkin.[1074] Github–Local

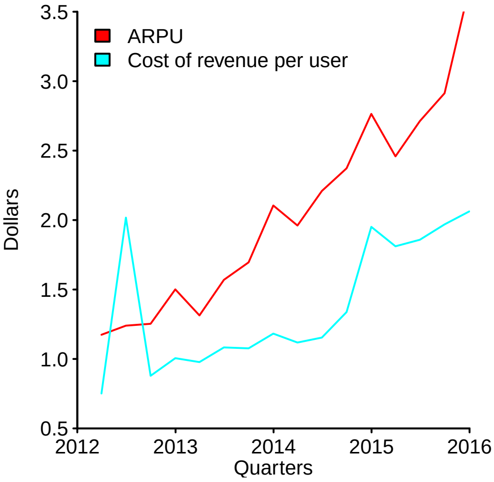


Figure 3.43: Facebook's ARPU and cost of revenue per user. Data from Facebook's 10-K filings.[560,561] Github–Local

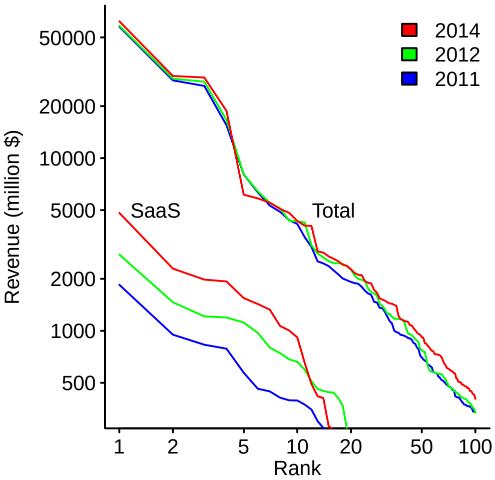


Figure 3.44: Top 100 software companies ranked by total revenue (in millions of dollars) and ranked by Software-as-a-Service revenue. Data from PwC.[1518–1520] Github–Local

by the models, and the data used, suggest that around 85% of users run pirated copies; see Github–economics/MPRA/. The Business Software Alliance calculates piracy rates by comparing the volume of software sales against an estimate of the number of computers in use, a method that has a high degree of uncertainty because of the many assumptions involved;[1478] see Github–economics/piracy$_{H}ICSS-2010.R$.

In some ecosystems (e.g., mobile) many applications are only used for a short period, after they have been installed; see fig 4.44.

In some markets most sales are closed just before the end of each yearly sales quarter, e.g., enterprise software. A study by Larkin[1074] investigated the impact of non-linear incentive schemes[xx] on the timing of deals closed by salespeople, whose primary income came from commission on the sales they closed. Accelerated commission schemes create an incentive for salespeople to book all sales in a single quarter.

Figure 3.42 shows the number of deals closed by week of the quarter, and the average agreed discount. Reasons for the significant peak in the number of deals closed at the end of the quarter include salespeople gaming the system to maximise commission and customers holding out for a better deal.

3.6.4 Managing customers as investments

Acquiring new customers can be very expensive, and it is important to maximise the revenue from those that are acquired.

What is the total value of a customer to a company?

If a customer makes regular payments of m, the *customer lifetime value* (CLV) is given by (assuming the payment is made at the end of the period; simply add m if payment occurs at the start of the period):

$$CLV = \frac{mr}{(1+i)} + \frac{mr^2}{(1+i)^2} + \frac{mr^3}{(1+i)^3} + \cdots \quad = m\frac{r}{1-r+i}\left[1-\left(\frac{r}{1+i}\right)^n\right]$$

where: r is the customer retention rate for the period, i the interest rate for the period, and n is the number of payment periods. As $n \to \infty$, this simplifies to: $CLV = m\frac{r}{1-r+i}$.

A person who uses a software system without paying for it may be valued as a product. Facebook values its users (whose eyeballs are the product that Facebook sells to its customers: advertisers) using ARPU,[xxi] defined as " . . . total revenue in a given geography during a given quarter, divided by the average of the number of MAUs in the geography at the beginning and end of the quarter." Figure 3.43 shows ARPU, and cost of revenue per user (the difference is one definition of profit, or loss).

In the business world, annual maintenance agreements are a common form of regular payment, another is the sale of new versions of the product to existing customers (often at a discount).

Before renewing their maintenance agreement, customers expect to see a worthwhile return on this investment. Possible benefits include new product features (or at least promises of these), and support with helping to fix issues encountered by the customer. Vendor's need to be able to regularly offer product improvements means it is not in their interest to include too many new features, or fix too many open issues, in each release; something always needs to be left for the next release.

3.6.5 Commons-based peer-production

The visibility of widely used open source applications, created by small groups of individuals, continues to inspire others to freely invest their energies creating or evolving software systems.

The quantity of open source software that is good enough for many commercial uses has made the support and maintenance of some applications a viable business model.[1622]

[xx] The percentage commission earned in a non-linear scheme depends on the total value of sales booked in the current quarter, increasing at specified points, e.g., a salesperson booking $250,000 in a quarter earns 2% commission, while a salesperson booking over $6 million earns a commission of 25%; that first $250,000 earns the first salesperson a commission of $5,000, while it earns the second salesperson $62,500.

[xxi] ARPU—Average Revenue Per User, MAU—Monthly Average Users.

Some commercial companies relicense software they have developed internally under an Open source license, and may spend significant amounts of money paying employees to work on open source software. The possible returns from making investments include:

- driving down the cost of complementary products (i.e., products used in association with the vendor's product, that are not substitutes that compete), this reduces the total cost to the customer, increasing demand, and making it more difficult for potential competitors to thrive.[242] Methods for driving down costs include supporting the development of free software that helps ensure there is a competitive market for the complementary product,
- giving software away to make money from the sale of the hardware that uses it,
- control of important functionality. For instance, Apple is the primary financial supporter of the LLVM compiler chain, which is licensed under the University of Illinois/NCSA Open source license; this license does not require contributors to supply the source code of any changes they make (unlike the GNU licensed GCC compilers). Over time LLVM has become an industrial strength compiler, making it the compiler of choice for vendors who don't want to release any code they develop.

The income stream for some applications comes from advertising, that runs during program execution. A study by Shekhar, Dietz and Wallach[1665] investigated advertising libraries used by 10,000 applications in the Android market, and the Amazon App Store, during 2012. Figure 3.45 shows the number of apps containing a given number of advertising libraries; line is fitted a Negative Binomial distribution.

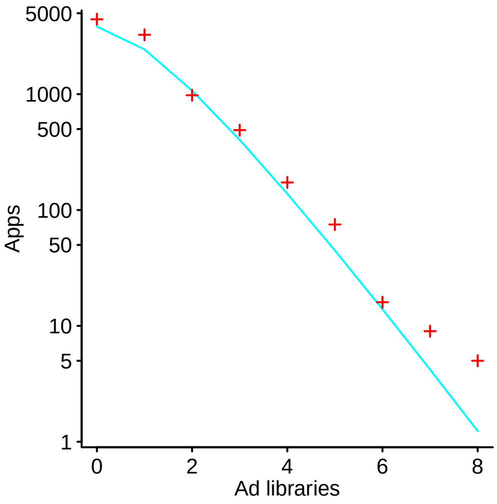


Figure 3.45: Number of applications in the Android market and Amazon App Store, during 2012, containing a given number of advertising libraries (line is a fitted Negative Binomial distribution). Data from Shekhar et al.[1665] Github–Local

Chapter 4

Ecosystems

4.1 Introduction

Customer demand motivates the supply of energy that drives software ecosystems.

Computer hardware is the terrain on which software systems have to survive, and the history of the capacity limits of this terrain has shaped the culture of those that live off it.

People and existing code are the carriers of software culture within an ecosystem; continual evolution of the terrain has imbued many software cultures with a frontier mentality spirit.

The size and complexity of software systems are limited by the amount of memory available on customer computers. The 16-bit microcomputers were limited to 64K of memory, large enough for a team of one or two developers to build a commercially viable application; the low cost of entry produced an explosion of applications. When customers acquired systems containing a 32-bit microprocessor and larger capacity memory devices, it became possible to sell applications containing an increasing number of features; teams of developers were required, increasing the cost of entry.

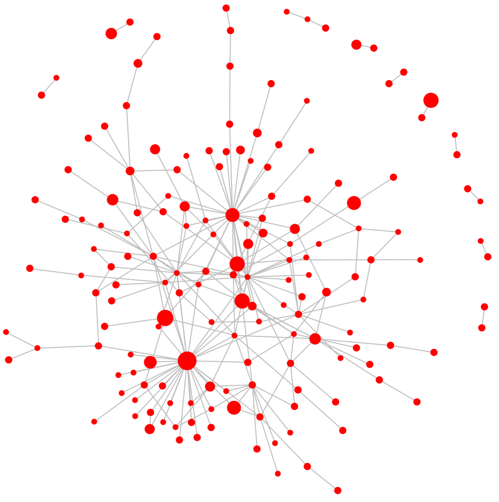


Figure 4.1: Connections between the 164 companies that have Apps included in the Microsoft Office365 Marketplace (Microsoft not included); vertex size is an indicator of the number of Apps a company has in the Marketplace. Data kindly provided by van Angeren.[1851] Github–Local

Figure 4.2 shows the amount of memory installed on systems running a SPEC benchmark on a given date, along with two fitted quantised regression lines. The lower line divides systems such that 95% are above it (i.e., contain more memory than systems below the line), the upper lines divides systems such that 50% are above it. The for 95% line, the amount of installed memory doubles every 845 days (it doubles every 825 days for the 50% line).

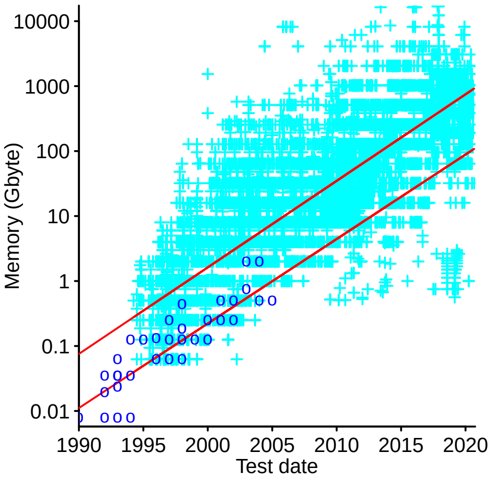


Figure 4.2: Amount of memory installed on systems running a SPEC benchmark on a given date; lines are fitted quantised regression models dividing systems into 50% above/below, and 95% above with 5% below.. Data from SPEC[1720] Github–Local

Software vendors have an interest in understanding the ecosystems in which they operate, want to estimate the expected lifetime of their products, estimate of the size of the potential market for their products and services, and to understand the community dynamics driving the exchange of resources. Governments have an interest in consumer well-being, which gives them an interest in the functioning of ecosystems with regard to the cooperation and competition between vendors operating within an ecosystem.

While software ecosystems share some of the characteristics of biological ecosystems, there are some significant differences, including:

- software ecosystem evolution is often Lamarkian [i], rather than Darwinian,[ii]
- software can be perfectly replicated with effectively zero cost, any evolutionary pressure comes from changes in the world in which the software is operated,
- like human genes, software needs people in order to replicate, but unlike genes software is not self-replicating; people replicate software when it provides something they want, and they are willing to invest in its replication, and perhaps even its ongoing maintenance,
- software has an unlimited lifetime, in theory, in practice many systems have dependencies on the ecosystem in which they run, e.g., third-party libraries and hardware functionality,

[i] A parent can pass on characteristics they acquired during their lifetime, to their offspring; modulo some amount of mixing from two parents, plus random noise.

[ii] Parents pass on the characteristics they were born with, to their offspring; modulo some amount of mixing from two parents, plus random noise.

- resource interactions are based on agreement, while in biological ecosystems creatures go to great lengths to avoid common resource interactions, e.g., being eaten.

Customer demand for software systems takes many forms, including: a means of reducing costs, creating new products to be sold for profit, tools that people can use to enhance their daily life, coercive pressure from regulators mandating certain practices,[1055] and experiencing the pleasure of creating something (e.g., writing open source, where the developer is the customer). Customer demand cannot always be supplied at a profit, and the demand for new products is often insufficient for them to be profitable.[iii] New products will diffuse into existing commercial ecosystems[1766] when they can be seen to provide worthwhile benefits.

The initial customer demand for computer systems followed patterns seen during the industrial revolution,[34] with established industries seeking to reduce their labor costs by investing in new technologies;[300, 1186] the creation of new industries based around the use of computers came later. The first computers were built from components manufactured for other purposes (e.g., radio equipment), as the market for computers grew, it became profitable to manufacture bespoke devices.

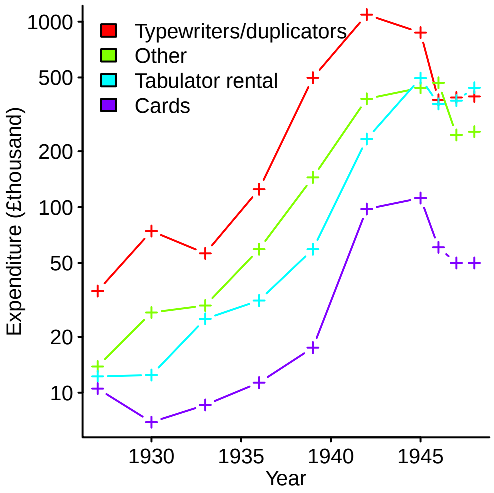


Figure 4.3: Yearly expenditure on punched cards, and tabulating equipment by the UK government. Data from Agar.[11] Github–Local

The data processing and numerical calculation ecosystems[733, 836] existed prior to the introduction of electronic computers; mechanical computers were used. Figure 4.3 shows UK government annual expenditure on punched cards, and mechanical data processing devices.

Customer demand for solutions to the problems now solved using software may have always existed; the availability of software solutions had to wait for the necessary hardware to become available at a price that could be supplied profitably;[276] see fig 1.1. Significant improvements in hardware performance meant that many customers found it cost effective to regularly replace existing computer systems. With many applications constrained by memory capacity, there was customer demand for greater memory capacity, incremental improvements in technology made it possible to manufacture greater capacity devices, and the expected sales volume made it attractive to invest in making the next incremental improvements happen; figure 4.4 shows the growth in world-wide DRAM shipments, by memory capacity shipped.

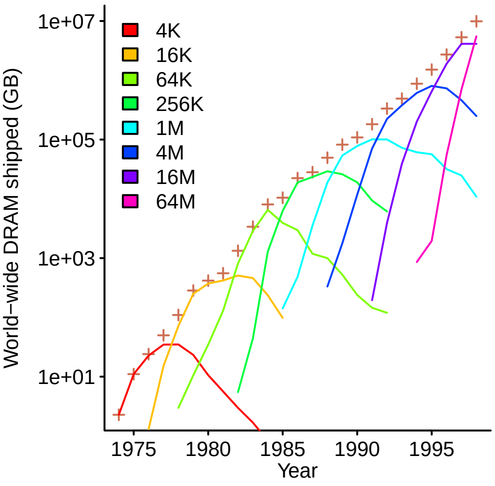


Figure 4.4: Total gigabytes of DRAM shipped world-wide in a given year, stratified by device capacity (in bits). Data from Victor et al.[1880] Github–Local

Who is the customer, and, which of their demands can be most profitably supplied? These tough questions are entrepreneurial and marketing problems,[326] and not the concern of this book.

The analysis in this book is oriented towards those involved in software development, rather than their customers, however it is difficult to completely ignore the dominant supplier of the energy (i.e., money) driving software ecosystems. The three ecosystems discussed in this chapter are oriented towards customers, vendors (i.e., companies engaged in economic activities, such as selling applications), and developers (i.e., people having careers in software development).

The firestorm of continual displacement of existing systems has kept the focus of practice on development of new systems, and major enhancements to existing systems. Post-firestorm, the focus of practice in software engineering is as an infrastructure discipline.[1837]

Software ecosystems have been rapidly evolving for many decades, and change is talked about as-if it were a defining characteristic, rather than a phase that ecosystems go through on the path to relative stasis. Change is so common that it has become blasé, the mantra has now become that existing ways of doing things must be disrupted. The real purpose of disruption is redirection of profits, from incumbents to those financing the development of systems intended to cause disruption. The fear of being disrupted by others is an incentive for incumbents to disrupt their own activities; only the paranoid survive[740] is a mantra for those striving to get ahead in a rapidly changing ecosystem.

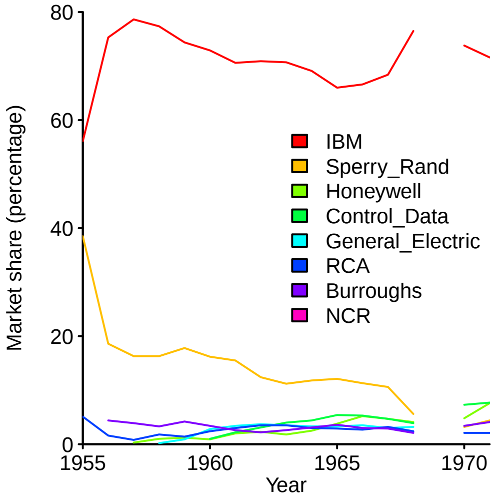


Figure 4.5: Computer installation market share of IBM, and its top seven competitors (known at the time as the seven dwarfs; no data is available for 1969). Data from Brock.[253] Github–Local

The first release of a commercial software project implements the clients' view of the world, from an earlier time. In a changing world, unimportant software systems have to adapted, if they are to continue to be used; important software systems force the world to adapt to be in a position to make use of them.

A study by van Angeren, Alves and Jansen[1851] investigated company and developer connections in several commercial application ecosystems. Figure 4.1 shows the connections

[iii] The first hand-held computer was introduced around 1989, and vendors regularly reintroduced such products, claiming that customer demand now existed. Mobile phones provided a benefit large enough that customers were willing to carry around an electronic device that required regular recharging; phones provided a platform for mobile computing that had a profitable technical solution.

between the 164 companies in the Microsoft Office365 Apps Marketplace (to be included, Apps have to meet platform added-value requirements).

Figure 4.5 illustrates how one company, IBM, dominated the first 30+ years of the computer industry. Figure 4.6 illustrates how, in new markets (mobile phones in this case) the introduction of a new platform can result in new market entrants replacing the existing dominant product ecosystems.

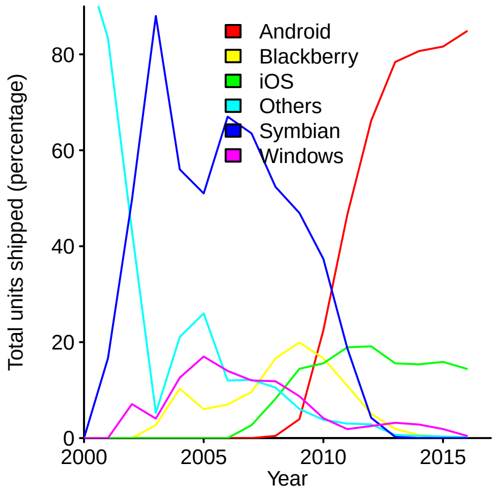


Figure 4.6: Mobile phone operating system shipments, as percentage of total per year. Data from Reimer[1551] (before 2007), and Gartner[648] (after 2006). Github–Local

Major geopolitical regions have distinct customer ecosystems, providing opportunities for region specific software systems to become established, e.g., Brazillian developer response to the imposition of strong trade barriers.[873]

The analysis of ecosystems can be approached from various perspectives: the population-community view of ecosystems is based on networks of interacting populations, while a process-functional view is based on the processing of resources across functional components. The choice of perspective used for ecosystem analysis may be driven by what data is available, e.g., it may be easier to measure populations than the resources used by a population.

4.1.1 Funding

The continued viability of a software ecosystem is dependent on funding (which may take the form of money or volunteer effort).

In a commercial environment the funding for development work usually comes from the products and services the organization provides. In a few cases investors fund startups with the intent of making a profit from the sale of the company (Martínez[1199] discusses working in a VC funded company).

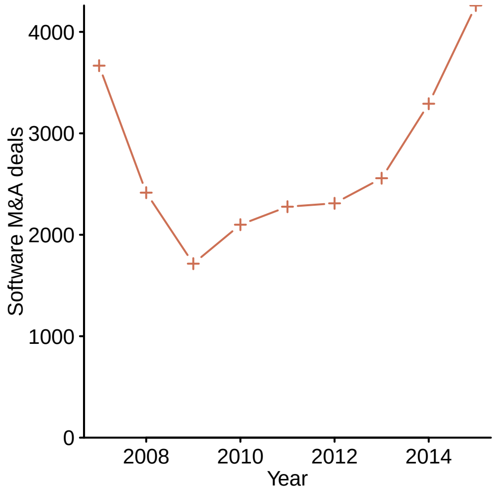


Figure 4.7: Reported number of worldwide software industry mergers and acquisitions (M&A), per year. Data from Solganick.[1713] Github–Local

Venture capital is a *hits business*, with a high failure rate, and most of the profit coming from a few huge successes. Exit strategies (extracting the profit from investments made in a company) used by VCs include: selling startups to a listed company (large companies may buy startups to acquire people with specific skills,[408] to remove potential future competitors, or because the acquired company has the potential to become a profit center; see figure 4.7), and an IPO (i.e., having the company's shares publicly traded on a stock exchange; between 2011 and 2015 the number of software company IPOs was in the teens, and for IT services and consulting companies the number was in single digits[1521]). Venture capitalists are typically paid a 2% management fee on committed capital, and a 20% profit-sharing structure;[1355] the VCs are making money, while those who invested in VCs over the last 20-years would have received greater returns by investing in a basket of stocks in the public exchanges.[1305]

Individuals who invest their own money in the early stage startups are sometimes called *Angel investors*. One study[1950] found that 30% of Angel investors lost their money, another 20% had a ROI of less than one, while 14% had a ROI greater than five; see Github–economics/SSRN-id1028592.R

Many Open source projects are funded by the private income of the individuals involved.

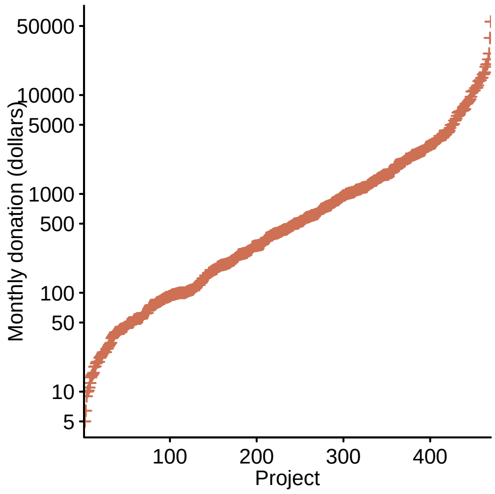


Figure 4.8: Average monthly donations received by 470 Github repositories using Patreon and OpenCollective. Data from Overney et al.[1411] Github–Local

A study by Overney, Meinicke, Kästner and Vasilescu[1411] investigated donations to Open Source projects. They found 25,885 projects on Github asking for donations, out of 78 million projects (as of 23 May 2019). Paypal was the most popular choice of donation platform, but does not reveal any information about donations received. Both Patreon and OpenCollective do reveal donation amounts, and 58% of the projects using these platforms received donations. Figure 4.8 shows the average monthly dollar amount received by Github projects having a Patreon or OpenCollective donate button in their README.md.

Advertising revenue as the primary income stream for software products is a relatively new phenomena, and vendor Ad libraries have been rapidly evolving.[18]

In the last 15 years over 100 non-profit software foundations have been created[893] to provide financial, legal and governance support for major open systems projects.

4.1.2 Hardware

The market for software systems is constrained by the number of customers with access to the necessary computer hardware, with the number of potential customers increasing as

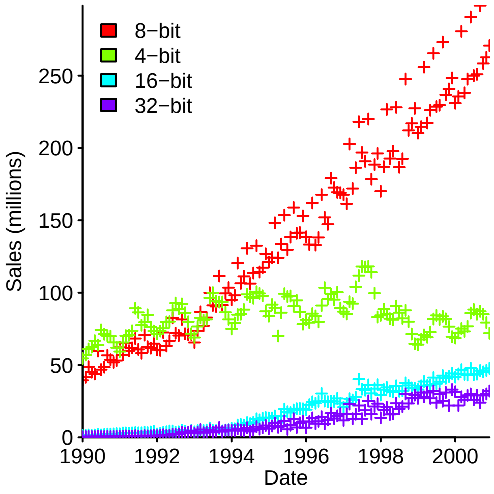


Figure 4.9: Monthly unit sales (in millions) of microprocessors having a given bus width. Data kindly provided by Turley.[1833] Github–Local

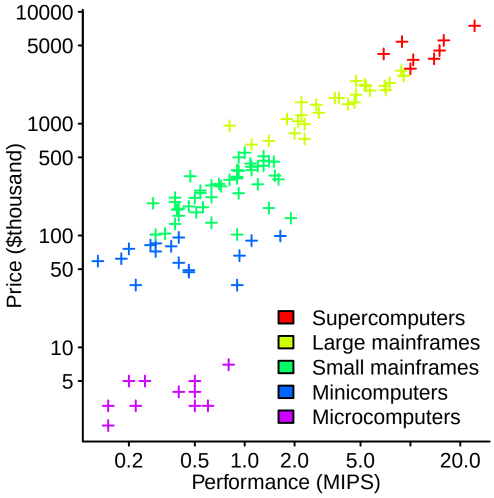


Figure 4.10: Performance, in MIPS, against price of 106 computer systems available in 1981. Data from Ein-Dor.[527] Github–Local

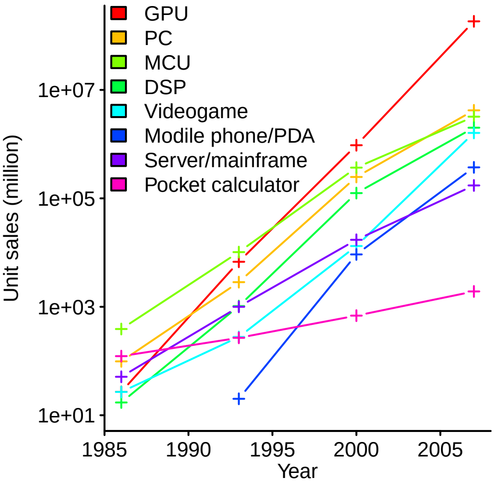


Figure 4.11: Total sales of various kinds of processors. Data from Hilbert et al.[818] Github–Local

the cost of the necessary computing hardware decreases. The functionality supported by software systems is constrained by the performance and capacity constraints of customer computing platforms; see fig 1.1 and fig 13.13 for illustrations of the cost of operations, and fig 1.2 for the cost of storage.

The general classification of computer systems[162] into mainframes, minicomputers[161] and microcomputers was primarily marketing driven,[iv] with each platform class occupying successively lower price points, targeting different kinds of customers (e.g., mainframes for large businesses,[v] minicomputers for technical and engineering companies, and micros for small companies and individuals). Supercomputing[163] (i.e., the fastest computers of the day, often used for scientific and engineering applications) is an example of a significant niche hardware ecosystem that has always existed. Embedded systems (where the computing aspect may be invisible to users) support several major ecosystems, with their own associations, conferences and trade press, but have not attracted as much attention from the software engineering research community as desktop systems. Figure 4.9 and Figure 4.11 show that in terms of microprocessor sales volume, the embedded systems market is significantly larger than what might be called the desktop computer market.

Figure 4.10 shows performance (in MIPS) against price for 106 computers, from different markets, in 1981. Microprocessors eventually achieved processor performance, and memory capacity, comparable to the more expensive classes of computers, at a lower cost; the lower cost hardware increased sales, which motivated companies to write applications for a wider customer base, which motivated the sale of more computers.[1071]

Manufacturers of computing systems once regularly introduce new product ranges containing cpus[147] that were incompatible with their existing product ranges. Even IBM, known for the compatibility of its 360/370 family of computers (first delivered in 1965, the IBM 360[1714] was the first backward compatible computer family: that is successive generations could run software that ran on earlier machines), continued to regularly introduced new systems based on cpus that were incompatible with existing systems.

What was the impact on software engineering ecosystems, of businesses having to regularly provide functionality on new computing platforms? At the individual level, a benefit for developers was an expanding job market, where they were always in demand. At the professional level the ever present threat of change makes codifying a substantial body of working practices a risky investment.

For its x86 family of processors, Intel made backwards compatibility a requirement[vi].[1297] An apparently insatiable market demand for faster processors, and large sales volume, created the incentive to continually make significant investments in building faster processors; see figure 4.11. The x86 family steadily increased market share,[1749] to eventually become dominant in the non-embedded computing market; one-by-one manufacturers of other processors ceased trading, and their computers have become museum pieces.

The market dominance of IBM hardware and associated software (50 to 70% of this market during 1969-1985;[503] see fig 1.5) is something that developers now learn through reading about the history of computing. However, the antitrust cases against IBM continue to influence how regulators think about how to deal with monopolies in the computer industry, and on how very large companies structure their businesses.[1432]

While the practices and techniques used during one hardware era (e.g., mainframes, minicomputers, microcomputers, the internet) might not carry over into later eras, they leave their mark on software culture, e.g., language standards written to handle more diverse hardware than exists today.[919] Also, each major new platform tends to be initially been populated with developers who are relatively inexperienced, and unfamiliar with what already exists, leading to many reinventions (under different names).

Changes in the relative performance of hardware components impact the characteristics of systems designed for maximum performance, which in turn impacts software design choices, which may take many years to catch up. For instance, the analysis of sorting algorithms once focused on the cost of comparing two items,[1016] but as this cost shrank in comparison to the time taken to load the values being compared, from memory, the analysis switched to focusing on cpu cache size.[1059]

[iv] The general characteristics of central processors and subsystems was very similar, and followed similar evolutionary paths because they were solving the same technical problems.

[v] Mainframes came with site planning manuals,[409] specifying minimum site requirements for items such as floor support loading, electrical power (in kilowatts) and room cooling.

[vi] To the extent of continuing to replicate faults present in earlier processors; see fig 6.3.

The greater the functional capacity of a computing system, the more power it consumes; see Github–benchmark/brl/brl.R. The energy consumed by computing devices is an important factor in some markets, from laptop and mobile phone battery usage,[125] compute infrastructure within a building[969] to the design of Warehouse-scale systems[134] (there are large energy efficiency gains to be had running software in the Cloud[1201]). The variation in power consumption between supposedly identical components can have a performance impact, i.e., devices reducing their clock rate to keep temperature within limits; see section 13.3.2.1.

The division between hardware and software can be very fuzzy; for instance, the hardware for Intel's Software Guard Extensions (SGX) instructions consists of software micro operations performed by lower level hardware.[402]

4.2 Evolution

Left untouched, software remains unchanged from the day it starts life; use does not cause it to wear out or break. However, the world in which software operates changes, and it is this changing world that reduces the utility of unchanged software. Software systems that have had minimal adaptation, to a substantially changed world,[195] are sometimes known as *legacy systems*.

The driving forces shaping software ecosystems have been rapidly evolving since digital computing began, e.g., hardware capability, customer demand, vendor dominance and software development fashion.

Competition between semiconductor vendors has resulted in a regular cycle of product updates; this update cycle has been choreographed by a roadmap published by the Semiconductor Industry Association.[1621] Cheaper/faster semiconductors drives cheaper/faster computers, generating the potential for businesses to update and compete more aggressively (than those selling or using slower computers, supporting less capacity). The hardware update cycle drives a Red Queen[129] treadmill, where businesses have to work to maintain their position, from fear that competitors will entice away their customers.

Figure 4.12 shows how wafer production revenue at the world's largest independent semiconductor foundry (TSMC) has migrated to smaller process technologies over time (upper), and how demand has shifted across major market segments (lower).

Companies introduce new products, such as new processors, and over time stop supplying them as they become unprofitable. Compiler vendors respond by adding support for new processors, and later reducing support costs by terminating support for those processors that have ceased to be actively targeted by developers. Figure 4.13 shows the survival curve for processor support in GCC, since 1999, and non-processor specific options.

Software evolves when existing source code is modified, or has new code added to it. Evolution requires people with the capacity to drive the changes, e.g., ability to make the changes, and/or fund others to do the work. Incentives for investing to adapt software, to a changed world, include:

- continuing to make money, from existing customers, through updates of an existing product. Updates may not fill any significant new customer need, but some markets have been trained, over decades of marketing, to believe that the latest version is always better than previous versions,
- needing to be competitive with other products,
- a potential new market opens up, and modifying an existing product is a cost effective way of entering this market, e.g., the creation of a new processor creates an opportunity for a compiler vendor to add support for a new cpu instruction set,
- the cost of adapting existing bespoke software is less than the cost of not changing it, i.e., the job the software did, before the world changed, still has to be done,
- software developers having a desire, and the time, to change existing code (the pleasure obtained from doing something interesting is a significant motivation for some of those involved in the production of software systems).

A product ecosystem sometimes experiences a period of rapid change; exponential improvement in product performance is not computer specific. Figure 4.14 shows the increase in the maximum speed of human vehicles on the surface of the Earth, and in the air, over time.

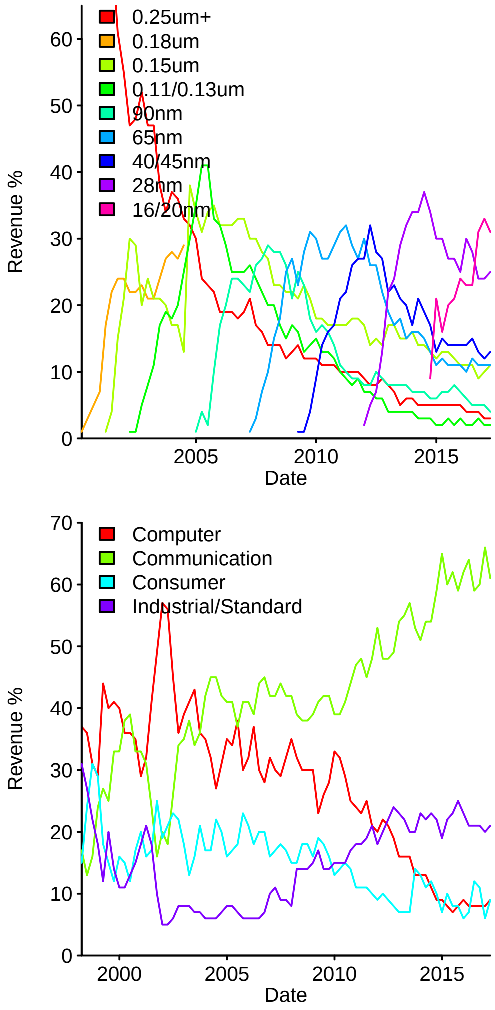


Figure 4.12: TSMC revenue from wafer production, as a percentage of total revenue, at various line widths. Data from TSMC.[1830] Github–Local

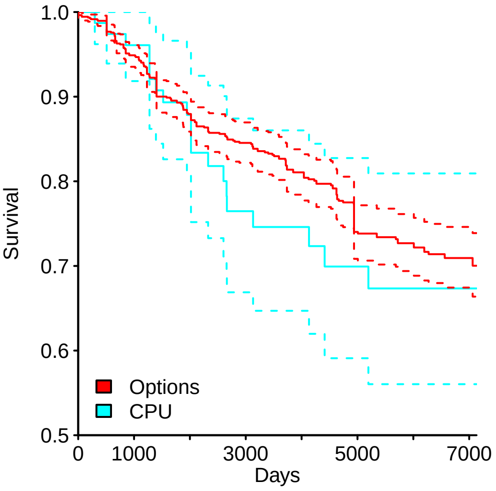


Figure 4.13: Survival curve for GCC's support for distinct cpus and non-processor specific compile-time options; with 95% confidence intervals. Data extracted from gcc website.[654] Github–Local

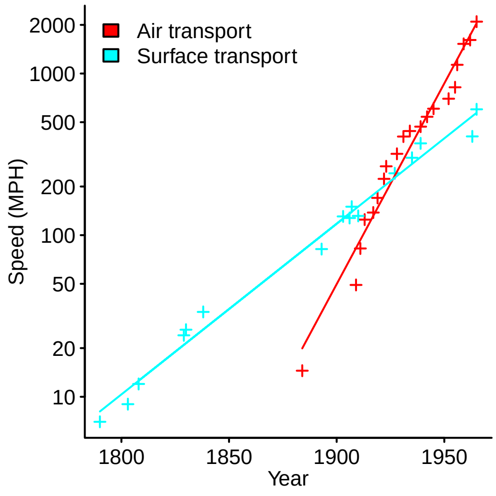


Figure 4.14: Maximum speed achieved by vehicles over the surface of the Earth, and in the air, over time. Data from Lienhard.[1129] Github–Local

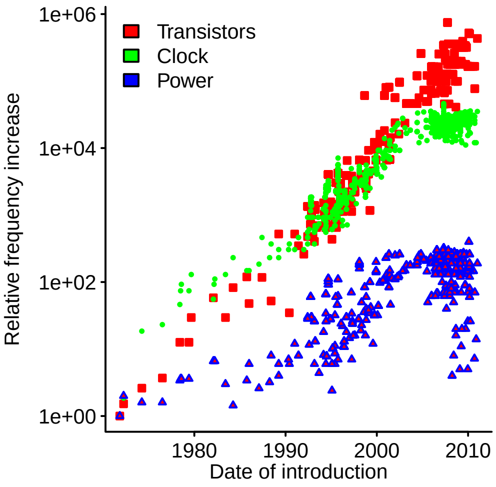


Figure 4.15: Number of transistors, frequency and SPEC performance of cpus when first launched. Data from Danowitz et al.[429] Github–Local

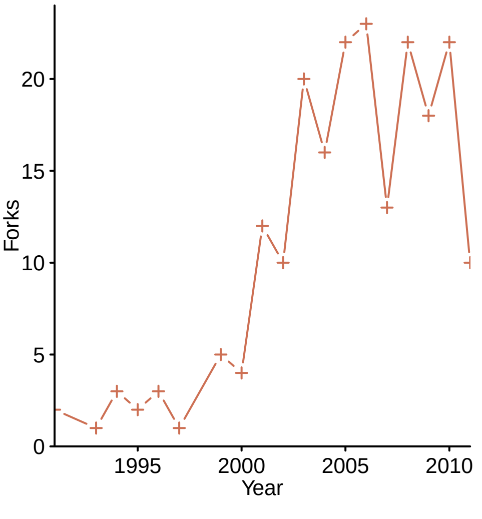


Figure 4.16: Number of major forks of projects per year, identified using Wikipedia during August 2011. Data from Robles et al.[1572] Github–Local

The evolution of a product may even stop, and restart sometime later. For instance, Microsoft stopped working on their web browser, Internet Explorer, reassigned and laid off the development staff; once this product started to lose significant market share, product development was restarted.

Ecosystem evolution is path dependent.[1743] Things are the way they are today because past decisions caused particular paths to be followed, driven by particular requirements, e.g., the QWERTY keyboard layout was designed to reduce jamming in the early mechanical typewriters,[431] not optimise typist performance (which requires a different layout[353]). The evolutionary paths followed in different locations can be different (e.g., evolution of religious[vii] beliefs[1962]).

Some evolutionary paths eventually reach a dead-end. For instance, NEC's PC-98 series of computers, first launched in 1979, achieved sufficient market share in Japan to compete with IBM compatible PCs, until the late 1990s (despite not being compatible with the effective standard computing platform outside of Japan).[1925]

Rapid semiconductor evolution appears to be coming to an end: an 18-month cycle for processor performance increases is now part of history. Figure 4.15 shows that while the number of transistors in a device has continued to increase, clock frequency has plateaued (the thermal energy generated by running at a higher frequency cannot be extracted fast enough to prevent devices destroying themselves).

When an ecosystem is evolving, rather than being in a steady state, analysis of measurements made at a particular point in time can produce a misleading picture. For instance, a snapshot of software systems currently being maintained may find that the majority have a maintenance/development cost ratio greater than five; however, the data suffers from survivorship bias, the actual ratio is closer to 0.8 (see fig 4.47, and Github–ecosystems/maint-dev-ratio.R). With many software ecosystems still in their growth phase, the rate of evolution may cause the patterns found in measurement data to change equally quickly.

4.2.1 Diversity

Diversity[viii] is important in biological ecology[319] because members of a habitat feed off each other (directly by eating, or indirectly by extracting nutrients from waste products). Software is not its own self-sustaining food web, its energy comes from the people willing to invest in it, and it nourishes those who choose to use it. The extent to which diversity metrics used in ecology are applicable to software, if at all, is unknown.

Studies[1764] have found that it was possible for organizations to make large savings by reducing software system diversity.

If software diversity is defined as variations on a theme, then what drives these variations and what are the themes?

Themes include: the software included as part of operating system distributions (e.g., a Linux distribution), the functionality supported by applications providing the same basic need (e.g., text editing), the functionality supported by successive releases of an application, by customised versions of the same release of an application, and the source code used to implement an algorithm (see fig 9.14).

A software system is *forked* when the applicable files are duplicated, and work progresses on this duplicate as a separate project from the original. The intent may be to later merge any changes into the original, to continue development independently of the original, some combination of these, or other possibilities (e.g., a company buying a license to adapt software for its internal use).

The *Fork* button on the main page of every project on GitHub is intended as a mechanism for developers, who need not be known to those involved in a project, to easily copy a project from which to learn, and perhaps make changes; possibly submitting any changes back to the original project, a process known as *fork and pull*. As of October 2013, there were 2,090,423 forks of the 2,253,893 non-forked repositories on GitHub.[906]

A study by Robles and González-Barahona,[1572] in 2011, attempted to identify all known significant forks; they identified 220 forked projects, based on a search of Wikipedia articles, followed by manual checking. Figure 4.16 suggests that after an initial spurt, the number of forks has not been growing at the same rate as the growth of open source projects.

[vii] There are over 33,830 denominations, with 150 having more than 1 million followers.[131]

[viii] Number of species and their abundance.

Software is created by people, and variations between people will produce variations in the software they write; other sources of variation include funding, customer requirements, and path dependencies present in the local ecosystem.

Reduced diversity is beneficial for some software vendors. The desktop market growth to dominance of Wintel[ix] reduced desktop computing platform diversity (i.e., the alternatives went out of business), which reduced support costs for software vendors (i.e., those still in business did not have to invest in supporting a wide diversity of platforms).

A study by Keil, Bennett, Bourgeois, Garcá-Peña, MacDonald, Meyer, Ramirez and Yguel[972] investigated the packages included in Debian distributions. Figure 4.17 shows a phylogenetic tree of 56 Debian derived distributions, based on the presence/absence of 50,708 packages in each distribution.

The BSD family of operating systems arose from forking during the early years of their development, and each fork has evolved as separate but closely related projects since the early-mid 1990s. Figure 9.22 illustrates how a few developers working on multiple BSD forks communicate bug fixes among themselves.

A study by Ray[1540] investigated the extent to which code created in one of NetBSD, OpenBSD or FreeBSD was ported to either of the other two versions, over a period of 18 years. Ported code not only originated in the most recently written code, but was taken from versions released many years earlier. Figure 4.18 shows the contribution made by 14 versions of NetBSD (versions are denoted by stepping through the colors of the rainbow) to 31 versions of OpenBSD; the contribution is measured as percentage of lines contained in all the lines changed in a given version.

Products are sometimes customised for specific market segments. Customization might be used to simplify product support, adapt to hardware resource constraints, and segmentation of markets as a sales tool.

Customization might occur during program start-up, with configuration information being read and used to control access to optional functionality, or at system build time, e.g., optional functionality is selected at compile time, creating a customised program.

Each customized version of a product can experience its own evolutionary pressures,[1430] and customization is a potential source of mistakes.

A study by Rothberg, Dintzner, Ziegler and Lohmann[1587] investigated the number of optional Linux features shared (i.e., supported) by a given number of processor architectures (for versions released between May 2011 and September 2013, during which the number of supported architectures grew from 24 to 30). Table 4.1 shows that the number of features only supported in one, two, three architectures, plus all supported architectures (and all but one, two and three).

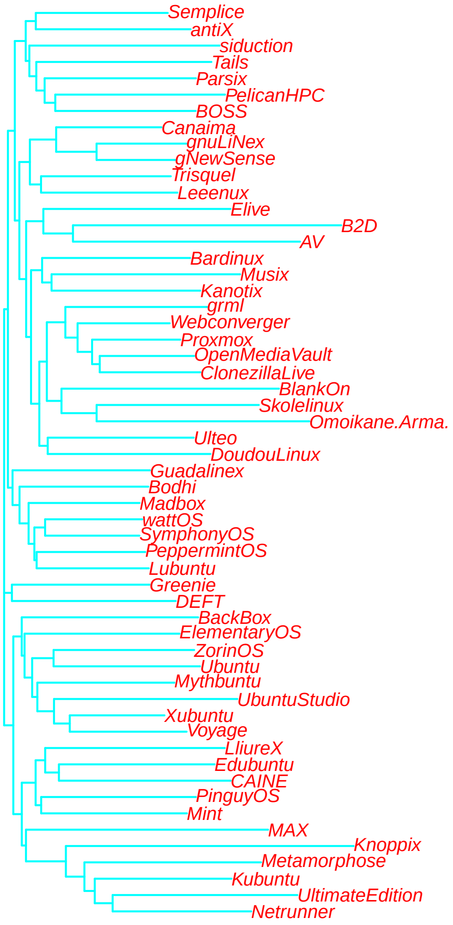


Figure 4.17: Phylogenetic tree of Debian derived distributions, based on which of 50,708 packages are included in each distribution. Data from Keil et al.[972] Github–Local

Version	1	2	3	All-3	All-2	All-1	All
2.6.39	3,989	182	50	2,293	944	1,189	2,617
3.0	3,990	183	53	2,345	968	1,211	2,637
3.1	4,026	184	52	2,440	968	1,155	2,667
3.2	4,028	181	57	1	2,788	512	4,054
3.3	4,077	180	51	1	2,837	512	4,133
3.4	4,087	183	51	1	2,907	520	4,184
3.5	4,129	179	50	2	3,001	520	4,265
3.6	4,158	184	51	2	3,098	527	4,298
3.7	4,139	183	50	1	3,173	539	4,384
3.8	4,148	178	35	3	3,269	548	4,399
3.9	4,269	177	36	3	3,403	581	4,413
3.10	4,280	173	35	3	3,447	577	4,460
3.11	4,270	178	33	2	0	0	8,654

Table 4.1: Number of distinct features, in Linux, shared across (i.e., supported) a given number of architectures (header row), for versions 2.6.39 to 3.11; **All** denotes all supported architectures (**All-1** is one less than **All**, etc), which is 24 in release 2.6.39, growing to 30 by release 3.9. Data from Rothberg et al.[1587]

The official Linux kernel distribution does not include all variants that exist in shipped products;[796] while manufacturers may make the source code of their changes publicly available, either they do not submit these changes to become part of the mainline distribution, or their submissions are not accepted into the official distribution (it would be a

[ix] Microsoft Windows coupled with Intel's x86 family of processors.

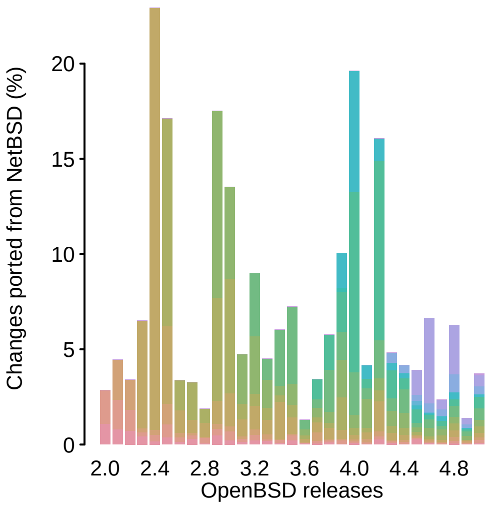


Figure 4.18: Percentage of code ported from NetBSD to various versions of OpenBSD, broken down by version of NetBSD in which it first occurred (denoted by incrementally changing color). Data kindly provided by Ray.[1540] Github–Local

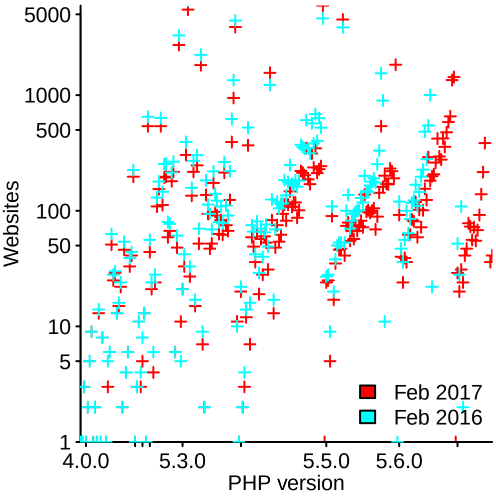


Figure 4.19: Number of websites running a given version of PHP on the first day of February, 2016 and 2017, ordered by PHP version number. Data kindly provided by Ruohonen.[1598] Github–Local

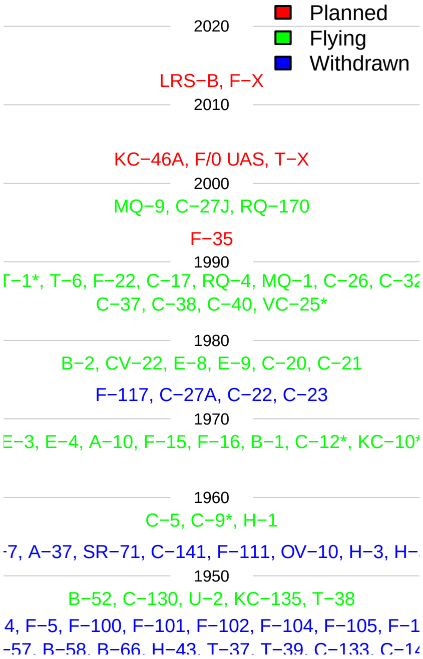


Figure 4.20: Decade in which newly designed US Air Force aircraft first flew, with colors indicating current operational status. Data from Echbeth el at.[519] Github–Local

heavy burden for the official Linux kernel distribution to include every modification made by a vendor shipping a modified kernel).

In general the number of optional features increases as the size of a program increases (see fig 7.34).

When a new version of a software system becomes available, there may be little, or no incentive for users to upgrade. As part of customer support, vendors may continue to provide support for previous versions, for some period after the release of a new version.

Widely used software systems may support their own ecosystem of third-party add-ons. For instance, Wordpress is written in PHP, and third-party add-ons are executed by the version of PHP installed on the server running Wordpress. The authors of these add-ons have to take into account the likely diversity in both the version of Wordpress and version of PHP used on any website.

A study by Ruohonen and Leppänen[1598] investigated the version of PHP available on more than 200K websites (using data from the httparchive[856]). Figure 4.19 shows the number of websites running a given version of PHP on the first day of February 2016 and February 2017; the x-axis is ordered by version number of the release (there is some intermingling of dates).

Hardware diversity (i.e., computer manufacturers offering a variety incompatible cpus and proprietary operating systems; with vendors seeking to create product moats) was once a major driver of software diversity; hardware issues are discussed in section 4.1.2.

4.2.2 Lifespan

The expected lifespan of a software system is of interest to those looking to invest in its creation or ongoing development. The investment may be vendor related (e.g., designing with an eye to future product sales) or customer related (e.g., integrating use of the software into business workflow). The analysis of data where the variable of interest is measured in terms of *time-to-event* is known as *survival analysis*; section 11.11 discusses the analysis of time-to-event data.

Governments may publicly express a preference for longer product lifetimes,[1291] because they are answerable to voters (the customers of products), not companies answerable to shareholders. In theory, a software system can only said to be dead when one or more of the files needed to run it ceases to exist. In practice, the lifespan of interest to those involved is the expected period of their involvement with the software system.

Factors affecting product lifetime include:

- in a volatile ecosystem there may be little or no incentive to maintain and support existing products. It may be more profitable to create new products without concern for compatibility with existing products,
- a product may not have sold well enough to make further investment worthwhile, or the market itself may shrink to the point where it is not economically viable to continue operating in it,
- while software does not wear out, it is intertwined with components of an ecosystem, and changes to third-party components that are depended upon can cause programs to malfunction, or even fail to execute. Changes in the outside world can cause unchanged software to stop working as intended,
- software depends on hardware for its interface to the world. Hardware wears out and breaks, which creates a post-sales revenue stream for vendors, from replacement sales. Manufacturing hardware requires an infrastructure, and specific components will have their own manufacturing requirements. The expected profit from future sales of a device may not make it worthwhile continuing to manufacture it, e.g., the sales volume of hard disks is decreasing, as NAND memory capacity performance, and cost per-bit improves,[615]
- users may decide not to upgrade because the software they are currently using is good enough. In some cases software lifespan may exceed the interval implied by the vendors official end-of-life support date.

The rate of product improvements can slow down for reasons that include: technology maturity, or a single vendor dominating the market (i.e., enjoys monopoly-like power). Figure 4.20 illustrates how the working life of jet-engined aircraft (invented in the same decade as computers) has increased over time. Figure 4.21 shows the mean age of installed mainframes at a time when the market was dominated by a single vendor (IBM), who decided product pricing and lifespan; the vendor could influence product lifespan to maximise their revenue.

Organizations and individuals create and support their own Linux distribution, often derived from one of the major distributions (e.g., Ubuntu was originally derived from Debian). Lundqvist and Rodic[1158] recorded the life and death of these distributions. Figure 4.22 shows the survival curve, stratified by the parent distribution.

The market share of the latest version of a software system typically grows to a peak, before declining as newer versions are released. Villard[1883] tracked the Android version usage over time. Figure 4.23 shows the percentage share of the Android market held by various releases, based on days since launch of each release.

A study by Tamai and Torimitsu[1791] investigated the lifespan of 95 software systems (which appear to be mostly in-house systems). Figure 4.24 shows (in red) the number of systems terminated after a given number of days, along with a fitted regression model of the form: $systems = ae^{b\times years}$ (with $b = -0.14$ for the mainframe software and $b = -0.24$ for Google's SaaS).

A website setup and maintained by Ogden[1386] records Google applications, services and hardware that have been terminated. Figure 4.24 shows (in blue/green) the number that have been terminated after a given number of days, along with a fitted regression model.

System half-life for the 1991 Japanese corporate mainframe data is almost 5-years; for the Google SaaS it is almost 2.9-years. Does the Japanese data represent a conservative upper bound, and the Google data a lower bound for a relatively new, large company finding out which of its acquired and in-house developed products are worth continuing to support?

Products with a short lifespan are not unique to software system. Between 1927 and 1960 at least 62 aircraft types were built and certified for commercial passenger transport[1461] (i.e., with seating capacity of at least four), and failed to be adopted by commercial airlines.

Communities of people have a lifespan. When the energy for the community comes directly from commercially activity, the community will continue to exist for as long as there are people willing to work for the money available, or they are not out competed by another community.[1387] When the energy comes directly from those active in the community, community existence is dependent on the continuing motivation and beliefs of its members.[829]

A study by Dunbar and Sosis[507] investigated human community sizes and lifetime. Figure 4.25 shows the number of founding members of 53, 19th century secular and religious utopian communities, along with the number of years they continued to exist, with loess regression lines.

4.2.3 Entering a market

Markets for particular classes of products (e.g., electric shavers, jet engines and video cassette recorders) and services evolve, with firms entering and existing.[767] Manufacturing markets have been found to broadly go through five stages, from birth to maturity.[12]

The invention of the Personal Computer triggered the creation of new markets, including people starting manufacturing companies to seek their fortune selling lower cost computers. Figure 4.26 shows how the growth and consolidation of PC manufacturers followed a similar trend to the companies started to manufacture automobiles (i.e., hundreds of companies started, but only a few survived).

Vendors want to control customer evolution, to the extent it involves product purchasing decisions. Evolution might be controlled by erecting barriers (the term *moat* is sometimes used) to either make it unprofitably for other companies to enter the market, or to increase the cost for customers seeking to switch suppliers.

- *economies of scale* (a supply side effect, a traditional barrier employed by hardware manufacturers): producing in large volumes allows firms to spread their fixed costs over more units, improving efficiency (requires that production be scalable in a way that allows existing facilities to produce more). Competing against a firm producing in volume requires a large capital investment to enter the market, otherwise the new entrant is at a cost disadvantage.

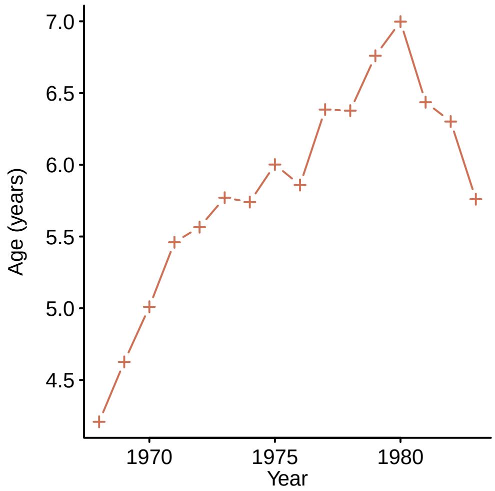


Figure 4.21: Mean age of installed mainframe computers, 1968-1983. Data from Greenstein.[729] Github–Local

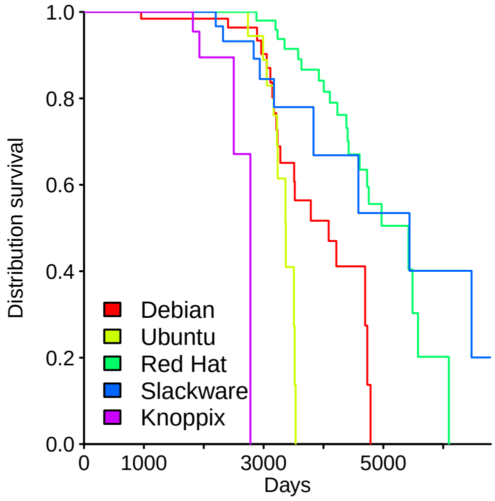


Figure 4.22: Survival curve of Linux distributions derived from five widely-used parent distributions (identified in legend). Data from Lundqvist et al.[1158] Github–Local

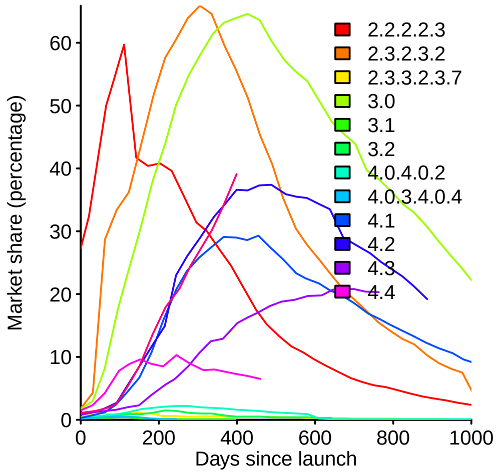


Figure 4.23: Percentage share of Android market, of a given release, by days since its launch. Data from Villard.[1883] Github–Local

With effectively zero replication costs, once created software systems do not require a large capital investment. A large percentage of the cost of software production is spent on people, who provide few opportunities for economies of scale (there may even be diseconomies of scale, e.g., communication overhead increases with head count),

- *network effects*[77] (a demand-side effect): are created when customers' willingness to buy from a supplier increases with the number of existing customers. A company entering a market that experiences large network effects, where they are competing against an established firm, has to make a large capital investment to offer incentives, so that a significant percentage of customers switch suppliers.

  Companies selling products that rely on network effects employ developer evangelists,[967] whose job includes creating a positive buzz around use of the product and providing feedback from third-party developers to in-house developers,
- *switching costs*:[569] a switching cost is an investment specific to the current supplier that must be duplicated to change to a new supplier, e.g., retraining staff and changes to procedures. The UK government paid £37.6 million transition costs to the winning bidder of the ASPIRE contract, with £5.7 million paid to the previous contract holder to facilitate changeover.[381]

  Information asymmetry can create switching costs by deterring competition. To encourage competition in bidding on the replacement ASPIRE contract (the incumbent had inside knowledge and was perceived to be strong) the UK government contributed £8.6 million towards bidders' costs.[381]

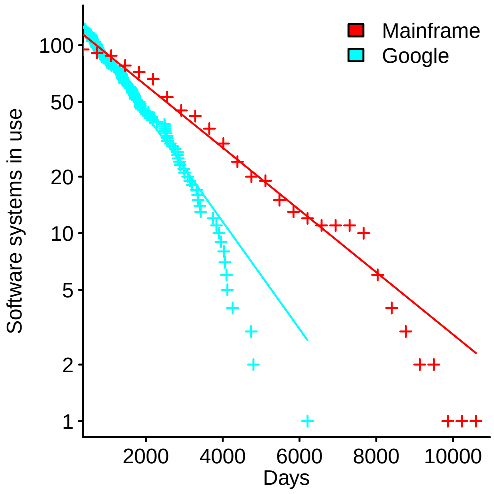


Figure 4.24: Number of software systems surviving for a given number of days and fitted regression models: Japanese mainframe software (red), Google software-as-a-service (blue). Data from: mainframe Tamai,[1791] Google's SaaS Ogden.[1386] Github–Local

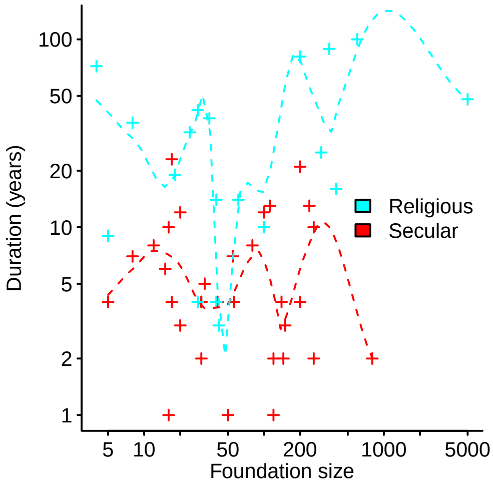


Figure 4.25: Size at foundation and lifetime of 32 secular and 19 religious 19th century American utopian communities; lines are fitted loess regression. Data from Dunbar et al.[507] Github–Local

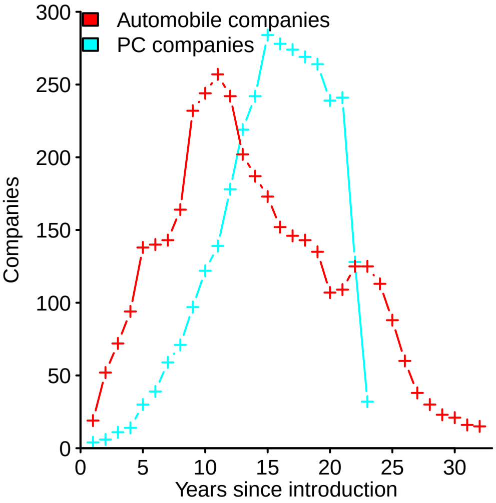


Figure 4.26: Number of US companies manufacturing automobiles and PCs, over the first 30-years of each industry. Data extracted from Mazzucato.[1211] Github–Local

Figure 4.27 shows the retail price of Model T Fords and the number sold, during the first nine years of the product.

A vendor with a profitable customer base, protected by products with high switching costs, is incentivised to prioritise the requirements of customers inside their walled garden. Income from existing customers sometimes causes vendors to ignore new developments that eventually lead to major market shifts,[570] that leave the vendor supporting a marooned customer based.

The competitive forces faced by companies include:[1489] rivalry between existing competitors, bargaining power of suppliers, bargaining power of buyers, possibility of new entrants, and possibility product being substituted by alternative products or services.

4.3 Population dynamics

Population dynamics[1223, 1371] is a fundamental component of ecosystems.

When the connections between members of an ecosystem are driven by some non-random processes, surprising properties can emerge, e.g., the *friends paradox*, where your friends have more friends, on average, than you do,[579] and the *majority illusion*, where views about a complete network are inferred from the views of local connections.[1104] In general, if there is a positive correlation, for some characteristic, between connected nodes in a network, a node's characteristic will be less than the average of the characteristic for the nodes connected to it.[541]

The choices made by individual members of an ecosystem can have a significant impact on population dynamics, which in turn can influence subsequent individual choices. Two techniques used to model population dynamics are: mathematics and simulation (covered in section 12.5).

The mathematical approach is often based on the use of differential equations: the behavior of the important variables are specified in one or more equations, which may be solved analytically or numerically.

One example is the evolution of the population of two distinct entities within a self-contained ecosystem. If, say, entities A and B have fitness a and b respectively, both have growth rate c, and an average fitness of ϕ, then the differential equations describing the evolution of their population size, x and y, over time are given by:[1371]

$$\dot{x} = ax^c - \phi x$$
$$\dot{y} = by^c - \phi y$$

Solving these equations shows that, when $c < 1$, both A and B can coexist, when $c = 1$, the entity with the higher fitness can invade and dominate an ecosystem (i.e., lower fitness eventually dies out), but when $c > 1$, an entity with high fitness cannot successfully invade an occupied ecosystem (i.e., greater fitness is not enough to displace an incumbent).

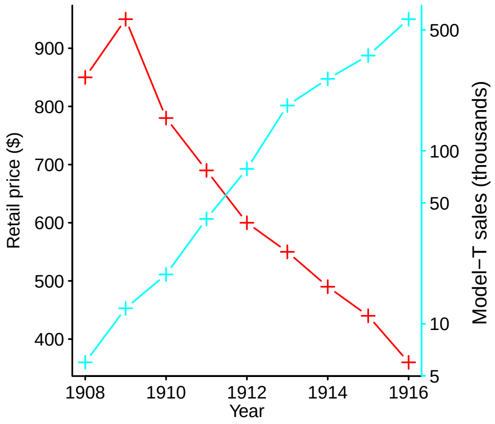


Figure 4.27: Retail prices of Model T Fords and sales volume. Data from Hounshell.[850] Github–Local

The mathematics approach has the advantage that, if the equations can be solved, the behavior of a system can be read from the equations that describe it (while simulations provide a collection of answers to specific initial conditions). The big disadvantage of the mathematical approach is that it may not be possible to solve the equations describing a real world problem.

The advantage of simulations is that they can handle most real world problems. The disadvantages of simulations include: difficulty of exploring the sensitivity of the results to changes in model parameters, difficulty of communicating the results to others, and computational cost; for instance, if 10^4 combinations of different model parameter values are needed to cover the possible behaviors in sufficient detail, with 10^3 simulations for each combination (needed to reliably estimate the mean outcome), then 10^7 simulation runs are needed, which at 1 second each is 116 cpu days.

When a product ecosystem experiences network effects, it is in vendors' interest to create what is known as a *virtuous circle*; encouraging third-party developers to sell their products within the ecosystem attracts more customers, which in turn attracts more developers, and so on.

Given two new technologies, say A and B, competing for customers in an existing market, what are the conditions under which one technology comes to dominate the market[76]?

Assume that at some random time, a customer has to make a decision to replace their existing technology, and there are two kinds of customer: R-agents perceive a greater benefit in using technology A (i.e., $a_R > b_R$), and S-agents perceive a greater benefit in using technology B (i.e., $a_S < b_S$); both technologies are subject to network effects, i.e., having other people using the same technology provides a benefit to everybody else using it.

Figure 4.28 illustrates the impact of the difference in customer adoption of two products (y-axis), with time along the x-axis; red-line is an example difference in the number of customers using products A and B. Between the blue/green lines, R and S-agents perceive a difference in product benefits; once the difference in the number of customers of each product crosses some threshold, both agents perceive the same product to have the greater benefit.

Table 4.2 shows the total benefit available to each kind of customer, from adopting one of the technologies; n_A and n_B are the number of users of A and B at the time a customer makes a decision, r and s are the benefits acrued to the respective agents from existing users (there are increasing returns when: $r > 0$ and $s > 0$, decreasing returns when: $r < 0$ and $s < 0$, and no existing user effect when: $r = 0$ and $s = 0$).

	Technology	**Technology**
	A	**B**
R-agent	$a_R + rn_A$	$b_R + rn_B$
S-agent	$a_S + sn_A$	$b_S + sn_B$

Table 4.2: Returns from choosing A or B, given previous technology adoptions by others. From Arthur.[77]

For increasing returns, lock-in of technology A occurs[77] (i.e., it provides the greater benefit for all future customers) when: $n_A(t) - n_B(t) > \frac{b_S - a_S}{s}$, where: $n_A(t)$ and $n_B(t)$ are the time dependent values of n.

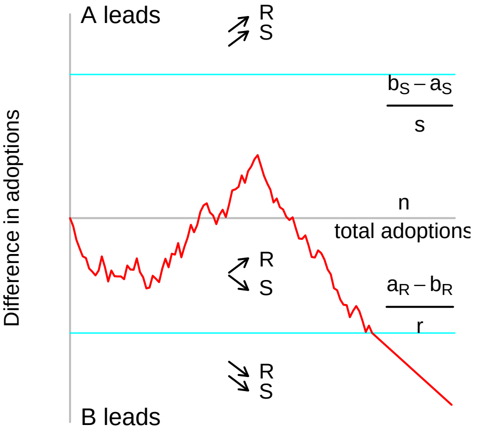


Figure 4.28: Example showing difference in number of customers using two products. Github–Local

The condition for lock-in of technology B is: $n_B(t) - n_A(t) > \frac{a_R - b_R}{r}$

Starting from a market with both technologies having the same number of customers, the probability that technology A eventually dominates is: $\frac{s(a_R - b_R)}{s(a_R - b_R) + r(b_S - a_S)}$ and technology B dominates with probability: $\frac{r(b_S - a_S)}{s(a_R - b_R) + r(b_S - a_S)}$

With decreasing returns, both technologies can coexist.

Governments have passed laws intended to ensure that the competitive process works as intended within commercially important ecosystems (in the US this is known as *antitrust law*, while elsewhere the term *competition law* is often used). In the US antitrust legal

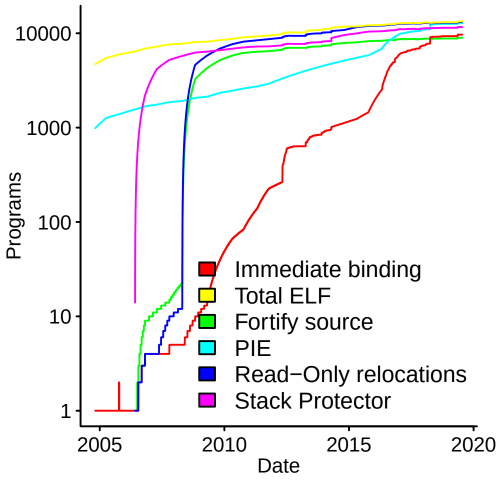


Figure 4.29: Number of programs in the Ubuntu AMD64 distribution shipped using a given security hardening technique (Total ELF is the number of ELF executables). Data from Cook.[390] Github–Local

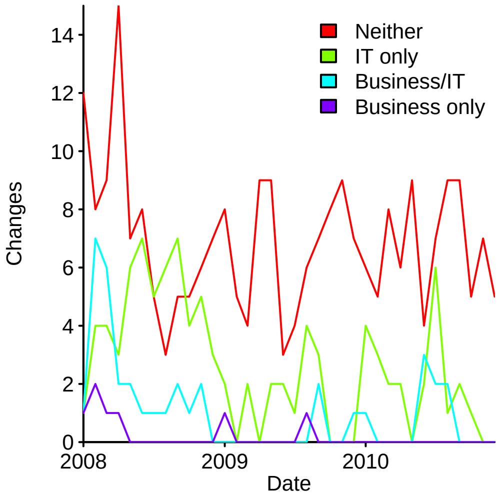


Figure 4.30: Number of process model change requests made in three years of a banking Customer registration project. Data kindly provided by Branco.[238] Github–Local

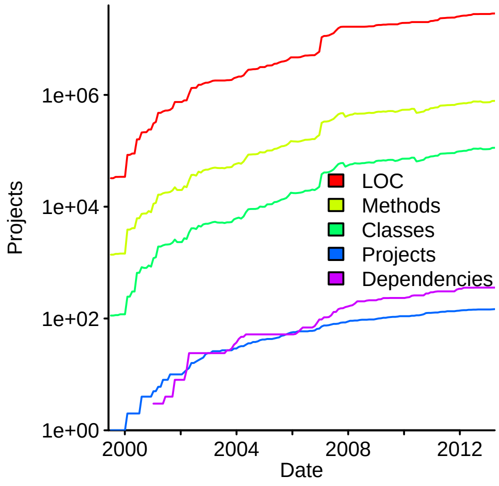


Figure 4.31: Growth in the number of projects within the Apache ecosystem, along with the amount of contained code. Data from Bavota et al.[148] Github–Local

thinking[981] in the 1960s had a market structure-based understanding of competition (i.e., courts blocked company mergers that they thought would lead to anticompetitive market structures). This shifted in the 1980s, with competition assessment based on the short-term interests of consumers (i.e., low consumer prices), not based on producers, or the health of the market as a whole.

The legal decisions and rules around ensuring that the competitive process operates in the commercial market for information are new and evolving.[1432]

In the UK and US it is not illegal for a company to have monopoly power within a market. It is abuse of a dominant market position that gets the attention of authorities; governments sometimes ask the courts to block a merger because they believe it would significantly reduce competition.[1639]

4.3.1 Growth processes

Population dynamics are sometimes modeled using a one-step at a time growth algorithm; one common example is preferential attachment, which is discussed in section 8.3.1.

The percentage of subpopulations present in a population can be path dependent. Consider the population of an urn containing one red and one black ball. The growth process iterates the following sequence (known as the *Polya urn process*): a ball is randomly drawn from the urn, this ball, along with another ball of the same color is placed back in the urn. After a many iterations, what is the percentage of red and black balls in the urn?

Polya proved that the percentage of red and black balls always converges to some, uniformly distributed, random value. Each converged value is the outcome of the particular sequence of (random) draws that occurred early in the iteration process; any converged percentage between zero and 100 can occur. Convergence also occurs when the process starts with an urn containing more than two balls, with each ball having a different color; see Github–ecosystems/Polya-urn.R.

The Polya urn process has been extended to include a variety of starting conditions (e.g., an urn containing multiple balls of the same color), and replacement rules (e.g., adding multiple balls of the same color, or balls having the other color). General techniques for calculating exact solutions to problems involving balls of two colors are available.[600]

The spread of software use may involve competition between vendors offering compatible systems (e.g., documents and spreadsheets created using Microsoft Office or OpenOffice[1585]), a system competing with an earlier version of itself (the Bass model is discussed in section 3.6.3), or the diffusion of software systems into new markets.[397]

Over time various techniques have been developed to make it more difficult for viruses to hijack programs; it takes time for effective techniques to be discovered, implemented and then used by developers. Figure 4.29 shows the growth in the number of programs in the Ubuntu AMD64 distribution that are hardened with a given security technique (Total ELF is the total number of programs).

Software change is driven by a mixture of business and technical factors. A study by Branco, Xiong, Czarnecki, Küster and Völzer[238] analysed over 1,000 change requests in 70 business process models of the Bank of Northeast of Brazil. Figure 4.30 shows the distribution of the 388 maintenance requests made during the first three years of the Customer Registration project. Over longer time-scales the rate of evolution of some systems exhibits a cyclic pattern.[1991]

Some very large systems grow through the integration of independently developed projects.

A study by Bavota, Canfora, Di Penta, Oliveto and Panichella[148] investigated the growth of projects in the Apache ecosystem. Figure 4.31 shows the growth in the number of projects, and coding constructs they contain. The number of dependencies between projects increases as the number of projects increases, along with the number of methods and/or classes; see Github–ecosystems/icsm2013_apache.R.

A later study by the same group[149] analyzed the 147 Java projects in the Apache ecosystem. Figure 4.32 shows, for each pair of projects, the percentage overlap of developers contributing to both projects (during 2013; white 0%, red 100%); the rows/columns have been reordered to show project pair clusters sharing a higher percentage of developers (see section 12.4.1).

4.3.2 Estimating population size

It may be impossible, or too costly, to observe all members of a population. Depending on the characteristics of the members of the population, it may be possible to estimate the number of members based on a sample; population estimates might also be derived from a theoretical analysis of the interaction between members.[1100]

In some cases it is possible to repurpose existing data. For instance, the U.S. Bureau of Labor Statistics (BLS) publishes national census information,[274] which includes information on citizens' primary occupation. The census occupation codes starting with 100, 101, 102, 104, and 106 have job titles that suggest a direct involvement in software development, with other codes suggesting some involvement. This information provides one means of estimating the number of software developers in the U.S. (one analysis[1473] of the 2016 data estimated between 3.4 million and 4.2 million).

When sampling from a population whose members have been categorized in some way (e.g., by species), two common kinds of sampling data are: *abundance data* which contains the number of individuals within each species in the sample, and *incidence data* giving a yes/no for the presence/absence of each species in the sample.

It is not possible to use incidence data to distinguish between different exponential order growth models,[1266] e.g., models based on a nonhomogeneous Poisson process. Modeling using incidence data requires samples from multiple sites (e.g., faults experienced within different companies, who use the software, tagged with location-id for each fault experience).

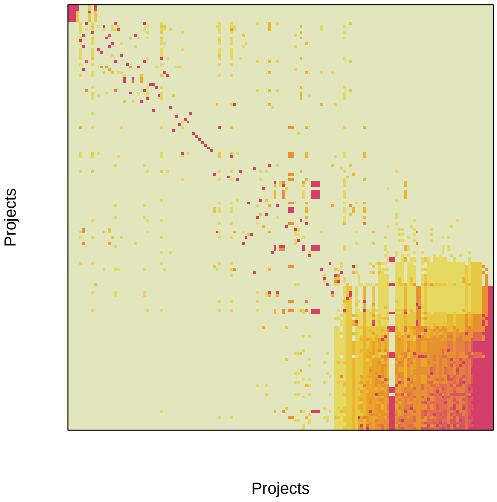


Figure 4.32: Percentage overlap of developers contributing, during 2013, to both of each pair of 147 Apache projects. Data kindly provided by Panichella.[149] Github–Local

4.3.2.1 Closed populations

In a closed population no members are added or removed, and the characteristics of the input distribution remain unchanged.

One technique for estimating (say) the number of fish in a lake, is to capture fish for a fixed amount of time, count and tag them, before returning the fish to the lake. After allowing the captured fish to disperse (but not so long that many fish are likely to have died or new fish born), the process is repeated, this time counting the number of tagged fish, and those captured for the first time (i.e., untagged).

The *Chapman estimator* is an unbiased, and more accurate estimate,[1604] than the simpler formula usually derived (i.e., $N = \frac{C_1 C_2}{C_{12}}$). Assuming that all fish have the same probability of being caught, and the probability of catching a fish is independent of catching any other fish:

$N = \dfrac{(C_1+1)(C_2+1)}{C_{12}+1} - 1$, where: N is the number of fish in the lake, C_1 the number of fish caught on the first attempt, C_2 the number caught on the second attempt, and C_{12} the number caught on both attempts.

Using the Chapman estimator to estimate (say) the number of issues not found during a code review assumes that those involved are equally skilled, invest the same amount of effort in the review process, and all issues are equally likely to be found. When reviewers have varying skills, invest varying amounts of effort, or the likelihood of detecting issues varies, the analysis is much more complicated. The `Rcapture` package supports the analysis of capture-recapture measurements where capture probability varies across items of interest, and those doing the capturing (also see the `VGAM` package).

The *Chao1* estimator[318] gives a lower bound on the number of members in a population, based on a count of each member captured; it assumes that each member of the population has its own probability of capture, that this probability is constant over all captures, and the population is sampled with replacement:

$$S_{est} \geq S_{obs} + \frac{n-1}{n}\frac{f_1^2}{2f_2}$$

where: S_{est} is the estimated number of unique members, S_{obs} the observed number of unique members, n the number of members in the sample, f_1 the number of items captured once, and f_2 the number of members captured twice.

If a population, containing N members, is sampled without replacement, the unseen member estimate added to S_{obs} becomes: $\dfrac{f_1^2}{\frac{n}{n-1}2f_2 + \frac{q}{1-q}f_1}$,[322] where: $q = \frac{n}{N}$.

Taking into account members occurring three and four times gives an improved lower bound.[347]

The ChaoSpecies function in the SpadeR package calculates species richness using a variety of models. The SpadeR and iNEXT packages contain functions for estimating and plotting species abundance data.

The number of additional members, m_g, that need to be sampled to be likely to encounter a given fraction, g, of all expected unique members in the population is:[320]

$$m_g \approx \frac{nf_1}{2f_2}\log\left[\frac{f_0}{(1-g)S_{est}}\right]$$

where: n is the number of members in the current sample and $f_0 = \frac{f_1^2}{2f_2}$. For $g = 1$, the following relationship needs to be solved for m: $2f_1\left(1+\frac{m}{n}\right) < e^{\frac{m}{n}\frac{2f_2}{f_1}}$

If m additional members are sampled, the expected number of unique members encountered is,[1666] assuming $m < n$ (when $n \le m \le n\log n$, more complicated analytic estimates are available[1403]):

$$S(n+m) = S_{obs} + f_0\left[1-\left(1-\frac{f_1}{nf_0+f_1}\right)^m\right]$$

If m is much less than n, this equation approximates to: $S(n+m) \approx S_{obs} + m\frac{f_1}{n}$.

The formula to calculate the number of unique members shared by two populations is based on the same ideas, and is somewhat involved.[1422]

When capture probabilities vary by time and individual item, the analysis is more difficult.[321]

Section 6.3.3 discusses the estimation of previously unseen fault experiences in a closed population.

4.3.2.2 Open populations

Evolving ecosystems contain open populations; in an open population existing members leave and new ones join, e.g., companies are started or enter a new market, and companies are acquired or go bankrupt. Estimates of the size of an open population that fail to take into account the impact of the time varying membership will not produce reliable results.

Most open population capture-recapture models are derived from the Cormack-Jolly-Seber (CJS) and Jolly-Seber (JS) models. CJS models estimate survival rate based on using an initial population of captured individuals and modeling subsequent recapture probabilities; CJS models do not estimate abundance. JS models estimate abundance and recruitment (new individuals entering the population). The openCR and Rcapture packages support the analysis of measurements of open populations.

Section 6.3.4 discusses the estimation of previously unseen fault experiences in an open population.

4.4 Organizations

Organizations contain ecosystems of people who pay for, use and create software systems.

4.4.1 Customers

Given that customers supply the energy that drives software ecosystems, the customers supplying the greatest amount of energy are likely to have the greatest influence on software engineering culture and practices (if somewhat indirectly). Culture is built up over time, and is path dependent, i.e., practices established in response to events early in the history of a software ecosystem may continue to be followed long after they have ceased to provide any benefits.

The military were the first customers for computers, and financed the production of one-off systems.[601] The first commercial computer, the Univac I, was introduced in 1951, and 46 were sold; the IBM 1401,[646] introduced in 1960, was the first computer to exceed one thousand installations, with an estimate 8,300 systems in use by July 1965.[1169] During the 1950s and 1960s the rapid rate of introduction of new computers created uncertainty, with many customers initially preferring to rent/lease equipment (management feared equipment obsolescence in a rapidly changing market,[1086] and had little desire to be responsible for maintenance[1052]), and vendors nearly always preferring to rent/lease rather than sell (a license agreement can restrict customers' ability to connect cheaper peripherals to equipment they do not own[472]). Figure 4.33 shows the shift from rental to purchase of computers by the US Federal Government.

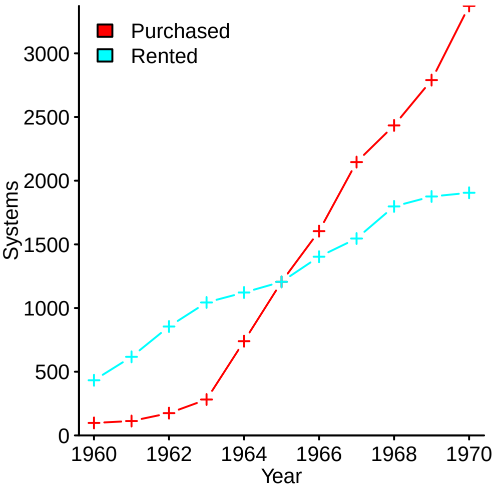


Figure 4.33: Total computer systems purchased and rented by the US Federal Government in the respective fiscal years ending June 30. Data from US Government General Accounting Office.[382] Github–Local

The U.S. Government was the largest customer for computing equipment during the 1950s and 1960s, and enacted laws to require vendors to supply equipment that conformed to various specified standards, e.g., the Brooks Act[1820] in 1965. The intent of these standards was to reduce costs, to the government, by limiting vendors ability to make use of proprietary interfaces that restricted competition.

The customers for these very expensive early computers operated within industries where large scale numeric calculations were the norm (e.g., banks, life insurance companies,[1972] and government departments[360]), and specific applications such as payroll in organizations employing thousands of people, each requiring a unique weekly or monthly payroll calculation.[728] Scientific and engineering calculations were a less profitable business.

Large organizations had problems that were large enough to warrant the high cost of buying and operating a computer.[1659] A computer services industry[1974] quickly sprang up (in the US during 1957, 32 of the 71 systems offered were from IBM[1168]), providing access to computers in central locations (often accessed via dial-in phone lines), with processing time rented by the hour.[84] Computing as a utility service for general use, like electricity, looked likely to become a viable business model.[1650] Figure 4.34 shows, for the US, service business revenue in comparison to revenue from system sales.

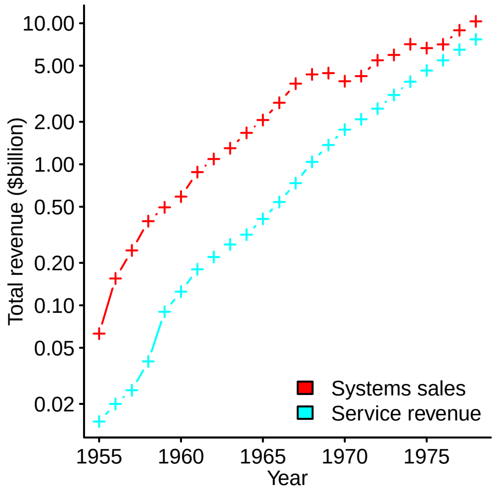


Figure 4.34: Total U.S. revenue from sale of computer systems and data processing service industry revenue. Data from Phister[1463] table II.1.20 and II.1.26. Github–Local

The introduction of microcomputers decimated the computer services business;[290] the cost of buying a microcomputer could be less than the monthly rental paid to a service bureau.

How much do customer ecosystems spend on software?

Figure 4.35 shows the total yearly amount spent by the UK's 21 industry sectors on their own software (which was reported by companies as fixed-assets; note: money may have been spent on software development, which for accounting purposed was not treated as a fixed-asset).

Customer influence over the trade-offs associated with software development ranges from total (at least in theory, when the software is being funded by a single customer[x]), to almost none (when the finished product is to be sold in volume to many customers). One trade-off is the reliability of software vs. its development cost (this issue is discussed in chapter 6). If the customer is operating in a rapidly changing environment, being first to market may have top priority. In a global market, variation in customer demand between countries[1132] has to be addressed.

Companies sometimes work together to create a product or service related ecosystem that services their needs. Competing to best address customer need results in product evolution; new requirements are created, existing requirements are modified or may become unnecessary.

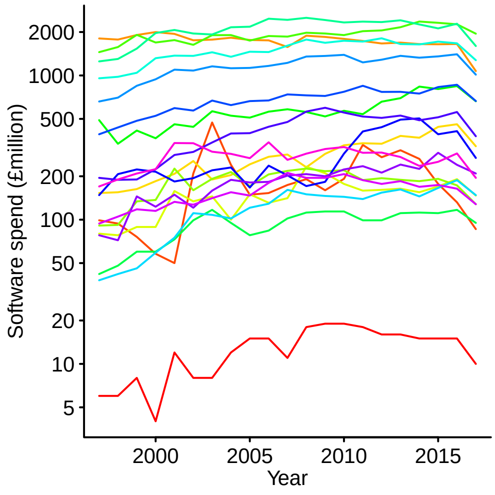


Figure 4.35: Total yearly spend on their own software by the 21 industry sectors in the UK, reported by companies as fixed-assets. Data from UK Office for National Statistics.[1385] Github–Local

A group of automotive businesses created AUTOSAR (Automotive Open System Architecture) with the aim of standardizing communication between the ECUs (Electronic Control Unit) supplied by different companies. A study by Motta[1300] investigated the evolution of the roughly 21,000 AUTOSAR requirements, over seven releases of the specification between 2009 and 2015. Figure 4.36 shows the cumulative addition, modification and deletion of software requirements during this period.

4.4.2 Culture

Cultures occur at multiple organizational levels. Countries and regions have cultures, communities and families have cultures, companies and departments have cultures, application domains and products have cultures, software development and programming

[x] Single customers may be large organizations paying millions to solve a major problem, or a single developer who enjoys creating software.

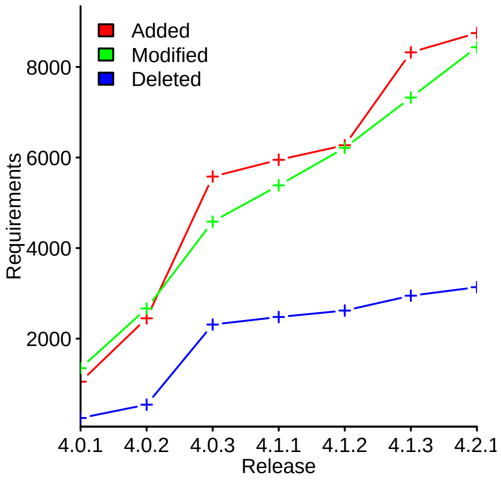


Figure 4.36: Cumulative number of software requirements added, modified and deleted, over successive releases, to the 11,000+ requirements present in release 4.0.0. Data kindly provided by Motta.[1300] Github–Local

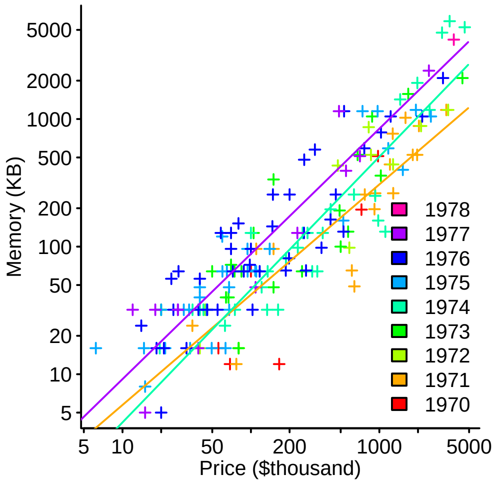


Figure 4.37: Typical memory capacity against cost of 167 different computer systems from 1970 to 1978; fitted regression lines are for 1971, 1974 and 1977. Data from Cale.[283] Github–Local

languages have cultures. Cultural differences are a source of confusion and misunderstanding, in dealings with other people.[1952]

Organizational routines are sometimes encapsulated as grammars of action, e.g., the procedures followed by a support group to solve customer issues.[1448]

The culture (e.g., learned behaviours, knowledge, beliefs and skills) embodied in each developer will influence their cognitive output. Shared culture provides benefits, such as, existing practices are understood and followed (e.g., the set of assumptions and expectations about how things are done), and using categorization to make generalizations from relatively small data sets (see section 2.5.4). Social learning is discussed in section 3.4.4.

Figure 4.37 shows what were considered to be typical memory capacities and costs, for 167 computer systems available from the major vendors, between 1970 and 1978; lines are fitted regression models for 1971, 1974 and 1977; see Github–ecosystems/CompWorld85.R. The performance and storage capacity limitations of early computers encouraged a software culture that eulogized the use of clever tricks, whose purpose was to minimise the amount of code and/or storage required to solve a problem.

A shared culture provides an aid for understanding others, but does not eliminate variability. When asked to name an object or action, people have been found to give a wide range of different names. A study by Furnas, Landauer, Gomez, and Dumais[630,631] described operations to subjects who were not domain experts (e.g., hypothetical text editing commands, categories in *Swap `n Sale* classified ads, keywords for recipes), and asked them to suggest a name for each operation. The results showed that the name selected by one subject was, on average, different from the name selected by 80% to 90% of the other subjects (one experiment included subjects who were domain experts, and the results for those subjects were consistent with this performance). The frequency of occurrence of the names chosen tended to follow a power law.

Metaphors[1058] are a figure of speech, which apply the features associated with one concept to another concept. For instance, concepts involving time are often expressed by native English speakers using a spatial metaphor. These metaphors take one of two forms—one in which time is stationary, and we move through it (e.g., "we're approaching the end of the year"); in the other, we are stationary and time moves toward us (e.g., "the time for action has arrived").

A study by Boroditsky[220] investigated subjects' selection of either the ego-moving, or the time-moving, frame of reference. Subjects first answered a questionnaire dealing with symmetrical objects moving to the left or to the right; the questions were intended to prime either an ego-moving or object-moving perspective. Subjects then read an ambiguous temporal sentence (e.g., "Next Wednesday's meeting has been moved forward two days") and were asked to give the new meeting day. The results found that 71% of subjects responded in a prime-consistent manner: of the subjects primed with the ego-moving frame, 73% thought the meeting was on Friday and 27% thought it was on Monday, and subjects primed with the object-moving frame showed the reverse bias (i.e., 31% and 69%).

Native Chinese speakers also use spatial metaphors to express time related concepts, but use the vertical axis rather than the horizontal axis used by native English speakers.

Cultural conventions can be domain specific. For instance, in the US politicians *run* for office, while in Spain and France they *walk*, and in Britain they *stand* for office. These metaphors may crop up as supposedly meaningful identifier names, e.g., `run_for`, `is_standing`.

Have the ecosystems inhabited by software changed so rapidly that cultural behaviors have not had time to spread and become widely adopted, before others take their place? Distinct development practices did evolve at the three early computer development sites in the UK.[288] Character encodings have been evolving since the 1870s,[596] with the introduction of the telegraph; early computer capacity restrictions drove further evolution,[1172] and the erosion of these restrictions has allowed a single encoding for all the World's characters[1844] to slowly spread.

Is English the lingua-franca of software development?

There has been a world-wide diffusion of software systems such as Unix, MS-DOS, and Microsoft Windows, along with the pre-internet era publicly available source code, such as X11, gcc, and Unix variants such as BSD; books and manuals have been written to teach software developers about this software. English is the language associated with these software systems and associated books, which has created incentives for developers in non-English speaking countries to attain good enough English proficiency.

When the interval between product updates is short, an investment in adapting products to non-English speaking markets is less likely to be economically worthwhile.

The world-wide spread of popular American culture is another vector for the use of English. In countries with relatively small populations it may not be economically viable to dub American/British films and TV programs, subtitles are a cheaper option (and these provide the opportunity for a population to learn English; see Github–ecosystems/Test_Vocab.txt).

The office suite LibreOffice was originally written by a German company, and many comments were written in German. A decision was made that all comments should be written in English. Figure 4.38 shows the decline in the number of comments written in German, contained in the LibreOffice source code.

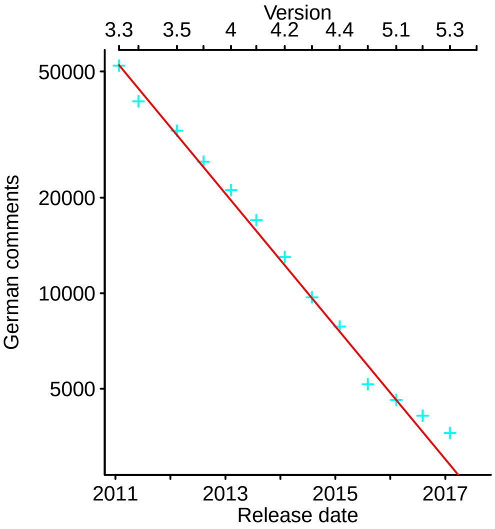


Figure 4.38: Estimated number of comments written in German, in the LibreOffice source code. Data from Meeks.[1240] Github–Local

Some companies, in non-English speaking countries, may require their staff to speak English at work.[1903]

In ALGEC,[838] a language invented in the Soviet Union, keywords can be written in a form that denotes their gender and number. For instance, Boolean can be written: логическое (neuter), логический (masculine), логическая (feminine) or логические (plural). The keyword for the "go to" token is to; something about Russian makes use of the word "go" unnecessary.

A fixation on a particular way of doing things is not limited to language conventions, examples can be found in mathematics. For instance, WEIRD people have been drilled in the use of a particular algorithm for dividing two numbers, which leads many to believe that the number representation used by the Romans caused them serious difficulties with division. Division of Roman numerals is straight-forward, if the appropriate algorithm is used.[352]

Ethnographic studies of engineering culture have investigated a large high-tech company in the mid-1980s,[1044] and an internet startup that had just IPO'ed.[1582]

Hacker culture are communities of practice, and encompass a variety of belief systems.[376]

ideology (a means of understanding a social order)...

4.4.3 Software vendors

A company is a legal entity that limits the liability of its owners. The duty of the directors of a company is to maximise the return on investment to shareholders. Commercial pressure is often to deliver results in the short term rather than taking a longer term view.[758]

The first software company, selling software consulting services, was founded in March 1955.[1036] Commercial software products, from third-parties, had to wait until computer usage became sufficiently widespread that a profitable market for them was considered likely to exist. One of the first third-party software products ran on the RCA 501, in 1964, and created program flowcharts.[910]

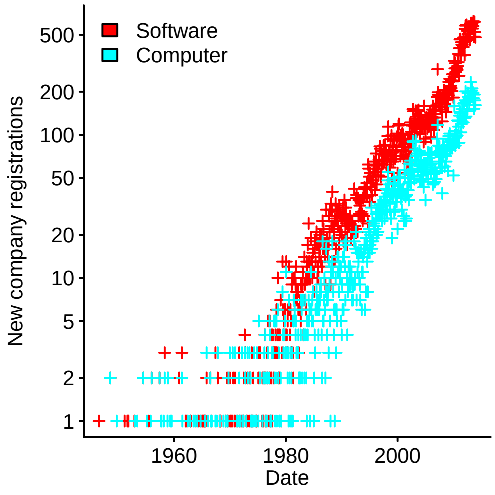


Figure 4.39: Number of new UK companies registered each month, whose SIC description includes the word software (45,422 entries) or computer (18,001 entries). Data extracted from OpenCorporates.[1400] Github–Local

The birth, growth, and death of companies[1012] is of economic interest to governments seeking to promote the economic well-being of their country. Company size, across countries and industries, roughly follows a Pareto distribution,[454] while company age has an exponential distribution;[367] most companies are small, and die young.

Figure 4.39 shows the number of UK companies registered each month having the word software (45,422 entries) or computer (18,001 entries) in their SIC description.

The development of a software system can involve a whole ecosystem of supply companies. For instance, the winner of a contract to develop a large military system often subcontracts-out work for many of the subsystems to different suppliers; each supplier using different computing platforms, languages and development tools (Baily et al[113] describes one such collection of subsystems). Subcontractors in turn might hire consultants, and specialist companies for training and recruitment.[368]

Companies may operate in specific customer industries (which themselves may be business ecosystems, e.g., the travel business), or a software ecosystem might be defined by the platform on which the software runs, e.g., software running on Microsoft Windows.

A study by Crooymans, Pradhan and Jansen[410] investigated the relationship connections between companies in a local software business network. Figure 4.40 shows the relationship links between these companies, with a few large companies and many smaller ones.[xi]

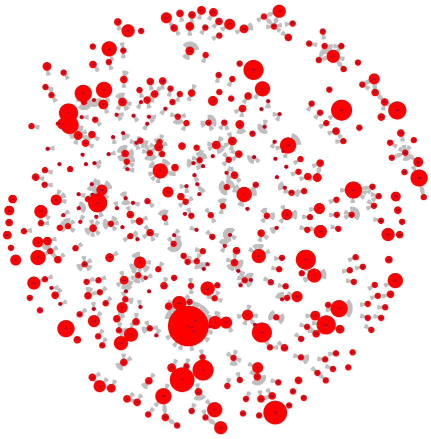


Figure 4.40: Connections between companies in a Dutch software business network. Data kindly provided by Crooymans.[410] Github–Local

4.4.4 Career paths

The availability of computing power has resulted in many new industries being created,[1245] but the stability needed to establish widely recognised software industry specific career paths has not yet occurred. A common engineering career path involves various levels of engineering seniority, taking on increasing management responsibilities, potentially followed by various levels of purely management activities;[35] the Peter principle may play a role in career progression.[172]

Coding was originally classified as a clerical activity (a significant under-appreciation of the cognitive abilities required), and the gender-based division of working roles prevalent during the invention of electronic computing assigned women to coding jobs; mens' role in the creation of software, during this period, included non-coding activities such as specifying the formulas to be used.[1130]

What are the skills and knowledge that employers seek in employees involved in software development, and what are the personal characteristics of people involved in such occupations?[xii]

The U.S., the Occupational Information Network (O*NET)[1397] maintains a database of information on almost 1,000 occupations, based on data collected from people currently doing the job.[xiii] The freely available data is aimed at people such as job seekers and HR staff, and includes information on the skills and knowledge needed to do the job, along with personal characteristics and experience of people doing the job.

Software companies employ people to perform a variety of jobs, including management,[1087] sales, marketing, engineering, Q/A, customer support, and internal support staff (e.g., secretarial). A study[900] of academic research units found that the ratio of support staff to academic staff was fitted by a power law having an exponent of around 1.30 (a hierarchical management model of one administrator per three sub-units produces an exponent of 1.26). Figure 3.4 shows that even in a software intensive organization around 87% of revenue is spent on non-software development activities.

Employment opportunities are the stepping stones of a career path, and personal preferences will result in some opportunities being considered more attractive than others. The popularity of particular software development activities[294] will have an impact on the caliber of potential employees available for selection by employers, e.g., maintenance activities are perceived as low status, an entry-level job for inexperienced staff to learn.[1782]

What roles might people having a career in software development fill, what are the common role transitions, how many people are involved, and how long do people inhabit any role?

Census information, and government employment statistics, are sources of data covering people who might be considered to be software developers; however, this data may include jobs that are not associated with software development. A study by Gilchrist and Weber[673] investigated the number of employed computer personnel in the US in 1970. The data for what was then known as *automatic data processing* included keypunching and computer operations personnel; approximately 30% of the 810,330 [xiv] people appear to have a claim to be software developers; see Github–ecosystems/50790641-II.R. This data does not include software development classified under R&D.

Figure 4.41 shows the number of people, stratified by age, employed in the 12 computer related U.S. Census Bureau occupation codes during 2014 (largest peak is the total).

People change, the companies they work for change, and software ecosystems evolve, which creates many of opportunities for people to change jobs.[438]

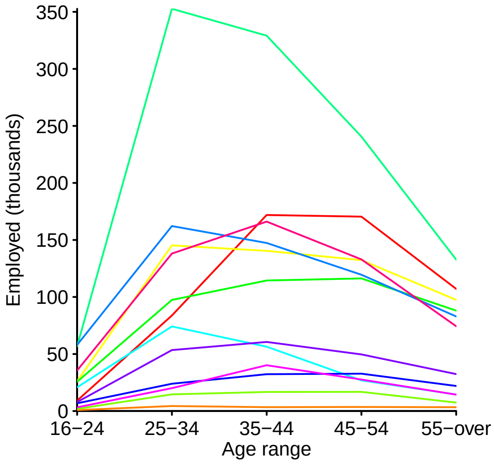


Figure 4.41: Number of people employed in the 12 computer occupation codes assigned by the U.S. Census Bureau during 2014, stratified by ages bands (main peak is the total, "Software developers, applications and system software" is the largest single percentage; see code for the identity of other occupation codes). Data from Beckhusen.[157] Github–Local

[xi] Teams of students decided which companies to interview, and so some clustering is the result of students using convenience sampling —email conversation with authors.

[xii] Government and industry interest in the availability engineering employees predates the availability of computers.[206]

[xiii] The O*NET website returns 20 matches for software developer occupations, at the time of writing; the U.S. Census Bureau maintains its own occupational classification.

[xiv] 127,491 working for the Federal government, 27,839 in state government extrapolated from data on 36 states and estimated 655,000 in private establishments.

When companies experience difficulties recruiting people with the required skills, salaries for the corresponding jobs are likely to increase. A growing number of companies ask employees to sign non-compete and no-poach agreements.[1742] An antitrust action was successfully prosecuted[575] against Adobe, Apple, Google, Intel, Intuit, and Pixar for mutually agreeing not to cold-call each other's employees (with a view to hiring them).

Startups have become a form of staff recruitment, with large companies buying startup companies to obtain a team with specific skills[1648] (known as *acqhiring* or *acqui-hiring*). One study[987] found that acqui-hired staff had higher turn-over compared to regular hires.

A study[209] of manufacturing industry found that family-friendly workplaces were associated with a higher-skilled workforce and female managers; based on the available data, there was no correlation with firm productivity.

One technique for motivating employees, when an organization is unable to increase their pay, is to give them an impressive job title. UK universities have created the job title *research software engineer*.[966]

A study by Joseph, Boh, Ang and Slaughter[947] investigated the job categories within the career paths of 500 people who had spent at least a year working in a technical IT role, based on data drawn from the National Longitudinal Survey of Youth. Figure 4.42 shows the seven career paths that (out of 13) included at least five years working in a technical IT role.

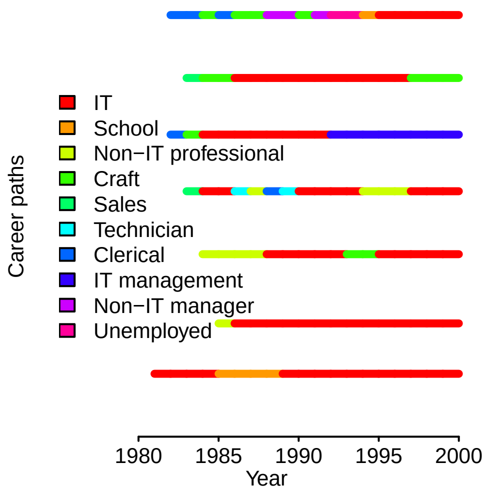


Figure 4.42: The job categories contained within the seven career paths in which people spent at least five years working in technical IT role. Data from Joseph et al.[947] Github–Local

There are opportunities for developers to make money outside formal employment.

Figure 4.43 shows a sorted list of the total amount earned by individuals through bug bounty programs. Both studies downloaded data available on the HackerOne website; the study by Zhao, Grossklags and Liu[1994] used data from November 2013 to August 2015, and the study by Maillart, Zhao, Grossklags and Chuang[1179] used data from March 2014 to February 2016.

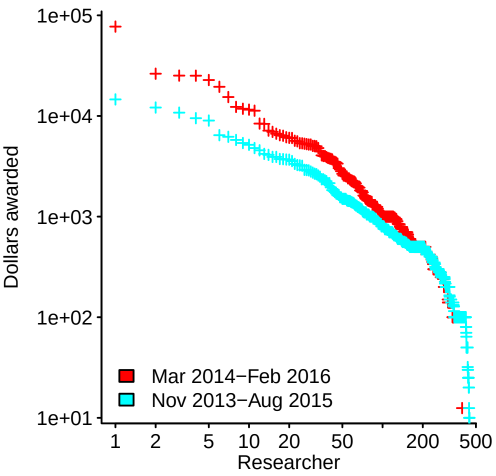


Figure 4.43: Sorted list of total amount awarded by bug bounties to individual researchers, based on two datasets downloaded from HackerOne. Data from Zhao et al[1994] and Maillart et al.[1179] Github–Local

4.5 Applications and Platforms

A successful software system attracts its own ecosystem of users and third-party developers. The accumulation of an ecosystem may be welcomed and encouraged by the successful owner, or it may be tolerated as a consequence of being successful.

Operating in a market where third-parties are tolerated by the apex vendor is a precarious business. Small companies may be able to eek out a living filling small niches that are not worth the time and effort for the apex vendor, or by having the agility to take advantage of short-term opportunities.

In a few product ecosystems the first release is everything, there is little ongoing market. Figure 4.44 shows the daily minutes spent using an App, installed from Apple's AppStore, against days since first used. This behavior was the case when people paid for games software to play on their phones; a shift to in-game purchases created an ongoing relationship.

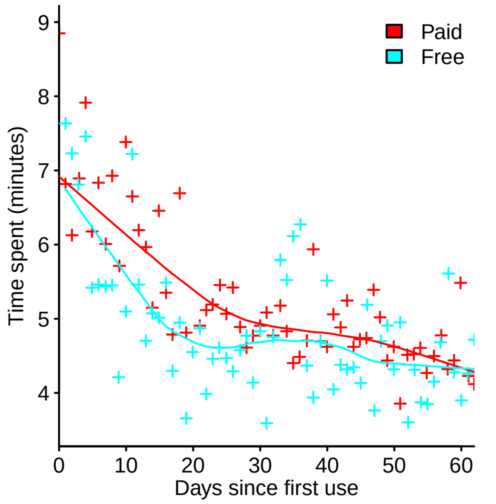


Figure 4.44: Daily minutes spent using an App, from Apple's AppStore (data from 2009); lines are a loess fit. Data extracted from Ansar.[61] Github–Local

In a rapidly evolving market, an ecosystem may effectively provide small companies within it, resources that exceed those available to much larger companies seeking to do everything themselves.

The rise of Open source has made it viable for substantial language ecosystems to flower, or rather substantial package ecosystems, with each based around a particular language. For practical purposes, a significant factor in language choice has become the quality and quantity of their ecosystem.

4.5.1 Platforms

Platform businesses[420] bring together producers and consumers, e.g., shopping malls link consumers and merchants, and newspapers connect readers (assumed to be potential customers) and advertisers; it is a *two-sided* market. Microsoft Windows is the poster child of a software platform.

Platforms create value by facilitating interactions between third-party producers and consumers; which differs from the value-chain model, which creates value by controlling a sequence of activities. An organization may act as an aggregator, collecting and making available items such as packages, source code, or job vacancies.

What has been called the Bill Gates line,[1812] provides one method of distinguishing between an aggregator and a platform: "A platform is when the economic value of everybody that uses it, exceeds the value of the company that creates it."

Platform owners aim to maximize the total value extracted from their ecosystem. In some cases this means subsidizing one kind of member to attract another kind (from which a greater value can be extracted). For instance, Microsoft once made development tools (e.g., compilers) freely available to attract developers, who wrote the applications that attracted end-users (there are significantly fewer developers than end-users, and any profit from selling compilers is correspondingly smaller).

Value-chains focus on customer value, and seek to maximize the lifetime value of individual customers, for products and services.

Organizations gain an advantage by controlling valuable assets that are difficult imitate. In the platform business the community, and the resources its members own and contribute is a valuable, hard to copy, asset. In a value-chain model these assets might be raw materials or intellectual property.

Building a platform requires convincing third-parties that it is worth joining, ecosystem governance is an important skill.

An organization that can create and own a de facto industry standard does not need to coordinate investments in creating the many enhancements and add-ons; an ecosystem of fragmented third-parties can work independently, they simply need to adhere to an established interface (what provides the coordination). The owner of a platform benefits from the competition between third-parties, who are striving to attract customers by creating desirable add-ons (which enhance the ecosystem, and fills the niches that are not worth the time and effort of the apex vendor).

Platform owners want customers to have a favorable perception of the third-party applications available on their platform. One way that platform owners can influence customer experience is to operate and police an App store that only allows approved Apps to be listed. A study[1899] of Google Play found that of the 1.5 million Apps listed in 2015, 0.79 million had been removed by 2017 (by which time 2.1 million Apps were listed).

Operating systems were the first software ecosystem, and OS vendors offered unique functionality to third-party developers in the hope they would use it extensively in the code they wrote (effectively increasing their own switching costs).

Vendors wanting to sell products on multiple operating systems have to decide whether to offer the same functionality across all versions of their product (i.e., not using functionality unique to one operating system to support functionality available to users of that OS), or to provide some functionality that varies between operating systems.

The term *middleware* is applied to software designed to make it easier to port applications across different operating systems; the Java language, and its associated virtual-machine, is perhaps the most well-known example of middleware.

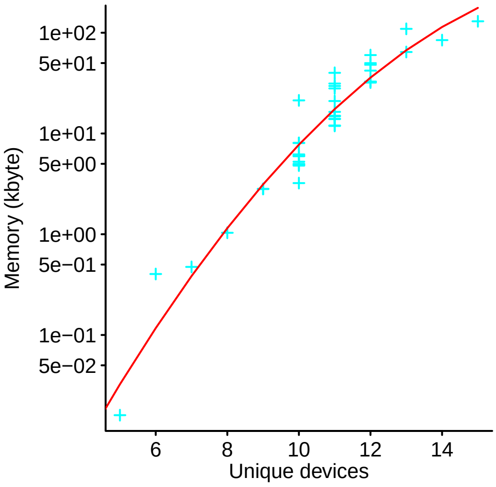


Figure 4.45: Size of 40 operating systems (Kbytes, measured in 1975) capable of controlling a given number of unique devices; line is a quadratic regression fit. Data from Elci.[531] Github–Local

Operating system vendors dislike middleware because it reduces switching costs. Microsoft, with successive versions of Microsoft Windows, was a dominant OS vendor during the 1990s and 2000s, and had an ecosystem control mechanism that others dubbed *embrace and extend*. Microsoft licensed Java, and added Windows specific functionality to its implementation, which then failed to pass the Java conformance test suite. Sun Microsystems (who owned Java at the time) took Microsoft to court and won;[1930] Microsoft then refused to ship a non-extended version of Java as part of Windows, Sun filed an antitrust case and won.[1301]

Small organizations seeking to create a platform might start out by building a base within an existing ecosystem. For instance, Zynga started as a games producer on the Facebook ecosystem, but then sought to migrate players onto its own platform (where it controlled the moneterization process).

4.5.2 Pounding the treadmill

Once a product ecosystem is established, investment by the apex vendor is focused on maintaining their position, and growing the ecosystem. A recurring trend is for software ecosystems to lose their business relevance, with the apex vendor remaining unchanged (see section 1.1).

Once up and running, some bespoke software systems become crucial assets for operating a business, and so companies have no choice but to pay whatever it takes to keep them running. From the vendors' perspective, maintenance is the least glamorous, but often the most profitable aspect of software systems; companies sometimes underbid to win a contract, and make their profit on maintenance activities (see chapter 5).

Over time, customers' work-flow molds itself around the workings of software products; an organization's established way of doing things evolves to take account of the behavior of the software it uses; staff training is a sunk cost. The cost of changing established practices, real or imaginary, is a moat that reduces the likelihood of customers switching to competing products; it is also a ball-and-chain for the vendor, in that it can limit product updates to those that do not generate costly changes for existing customers. At some point the profit potential of new customers may outweigh that of existing customers, resulting in a product update that requires existing customers to make a costly investment before they can adopt the new release.

Pressure to innovate comes from the economic benefits of having the installed base upgrade. Continual innovation avoids the saturation of demand, and makes it difficult for potential competitors to create viable alternatives.

Commercial products evolve when vendors believe that investing in updates is economically worthwhile. Updated versions of a product provide justification for asking customers to pay maintenance or upgrade fees, and in a competitive market work to maintain, or improve, market position, and address changing customer demand; product updates also signal to potential customers that the vendor has not abandoned the product (unfavourable publicity about failings in an existing product can deter potential new customers).

Product version numbers can be used to signal different kinds of information, such as which upgrades are available under a licensing agreement (e.g., updates with changes to the minor version number are included), and as a form of marketing to potential customers (who might view higher numbers as a sign of maturity). The release schedule and version of some systems is sufficiently stable that a reasonably accurate regression model can be fitted;[1864] see Github–regression/release_info/cocoon_mod.R.

The regular significant improvement in Intel cpu performance (see fig 4.15), starting in the last 1980s, became something that was factored into software system development, e.g., future performance improvements could be used as a reason for not investing much effort in tuning the performance of the next product release.

Files created by users of a product are a source of customer switching costs (e.g., cost of conversion), and vendor ball-and-chain (e.g., the cost of continuing to support files created by earlier versions). File written using a given format can have very long lifetime; a common vendor strategy is to continue supporting the ability to read older formats, but only support the writing of more recent formats. A study by Jackson[895] investigated pdf files created using a given version of the pdf specification. Figure 4.46 shows the total number of pdf files created using a given version of the pdf specification, available on websites having a .uk web domain between 1996 and 2010 (different pdf viewers do not always consistently generate the same visual display from the same pdf file[1038]).

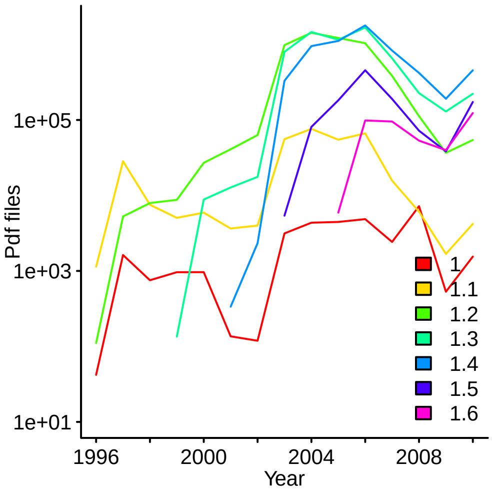


Figure 4.46: Number of pdf files created using a given version of the portable document format appearing on sites having a .uk web address between 1996 and 2010. Data from Jackson.[895] Github–Local

A few software systems have existed for many decades, and are expected to last many more decades, e.g., simulation of nuclear weapon performance[1492] (the nuclear test-ban treaty prohibits complete device testing).

What are the costs associated with maintaining a software system?

A study by Dunn[510] investigated the development and maintenance costs of 158 software systems developed by IBM (total costs over the first five years); some of these contained a significant percentage of COTS components. The systems varied in size from 34 to 44,070 man-hours of development effort, and involved from 21 to 78,121 man-hours of maintenance. Figure 4.47 shows the ratio of development to average annual maintenance cost. The data is for systems at a single point in time, i.e., 5-years. Modeling, using expected system lifetime, finds that the mean total maintenance to development cost ratio is less than one; see Github–ecosystems/maint-dev-ratio.R. The correlation between development and maintenance man-hours is 0.5 (0.38-0.63 is the 95% confidence interval); see Github–economics/maint-dev-cost-cor.R.

Hedonism funded software systems continue to change for as long as those involved continue to enjoy the experience.

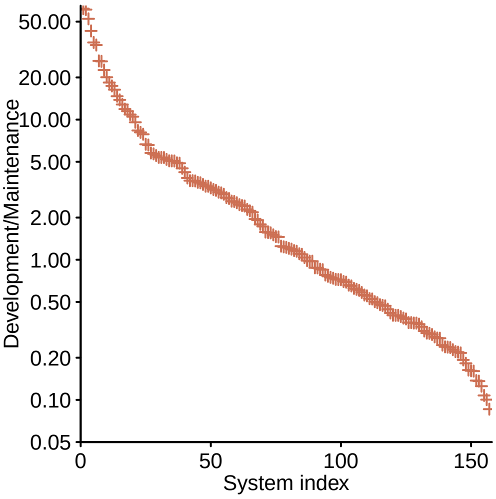


Figure 4.47: Ratio of development costs to average annual maintenance costs (over 5-years) for 158 IBM software systems sorted by size; curve is a beta distribution fitted to the data (in red). Data from Dunn.[510] Github–Local

The issues around fixing reported faults during maintenance are discussed in chapter 6. In safety critical applications the impact of changes during maintenance has to be thought through;[444] this issue is discussed in chapter 6.

4.5.3 Users' computers

A software system has to be usable within the constraints of the users' hardware, and the particular libraries installed on their computer.

The days when customers bought their first computer to run an application are long gone. Except for specialist applications[1077] and general hardware updates, it is not usually cost effective for customers to invest in new computing hardware specifically to run an application. Information on the characteristics of existing customer hardware is crucial because software that cannot be executed with a reasonable performance in the customer environment will not succeed (figure 8.27 shows the variation in basic hardware capacity of desktop systems).

The distribution of process execution times often has the form of either power law or an exponential.[577] In large computer installations, *workload analysis*,[577] analyzing computer usage and balancing competing requirements to maximise throughput, may employ teams in each time zone. Figure 4.48 shows the distribution of the execution time of 184,612 processes running on a 1995 era Unix computer.

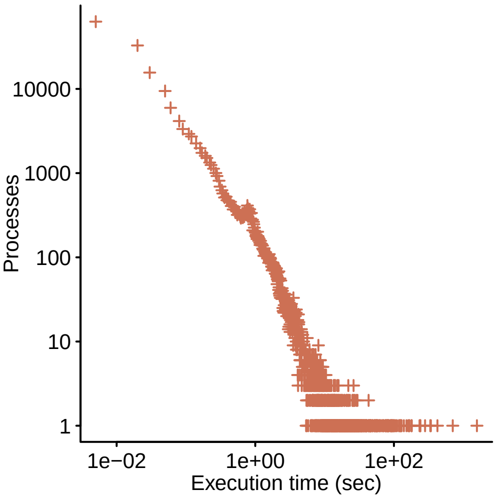


Figure 4.48: Number of Unix processes executing for a given number of seconds, on a 1995 era computer. Data from Harchol-Balter et al.[769] Github–Local

Obtaining solutions to a few problems is sufficiently important that specialist computers are built to run the applications, e.g., *super computers* (see Github–Rlang/Top500.R), and bespoke hardware to solve one problem.[1661]

4.6 Software development

Most computer hardware is brought because it is needed to run application software,[xv] i.e., software development is a niche activity.

Computers were originally delivered to customers as bare-metal (many early computers were designed and built by established electronics companies[641]); every customer wrote their own software, sometimes obtaining code from other users.[xvi] As experience and customers accumulated,[104] vendors learned about the kinds of functionality that was useful across their customer base.[271] Supplying basic software functionality needed by customers decreased computer ownership costs, and increased the number of potential customers.

Figure 4.49 shows how the amount of code shipped with IBM computers increased over time. Applications need to receive input from the outside world, and produce output in a form acceptable to their users; interfacing to many different peripherals is a major driver of OS growth, see figure 4.45.

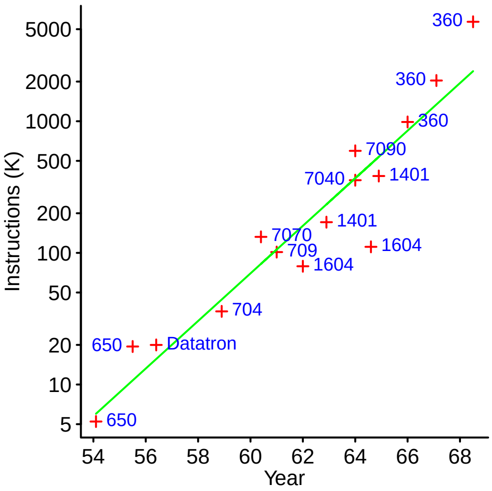


Figure 4.49: Total instructions contained in the software shipped with various models of IBM computer, plus Datatron from Burroughs; line is a fitted regression of the form: $Instructions \propto e^{0.4Year}$. Data extracted from Naur et al.[1337] Github–Local

All software for the early computers was written using machine code, which made it specific to one particular kind of computer. New computers were constantly being introduced, each having different characteristics (in particular machine instructions).[203, 1236, 1921, 1922]

Machine independent programming languages (e.g., Fortran in 1954[103] and Cobol in 1961[167]) were created with the aim of reducing costs. The cost reduction came through reducing the dependency of program source code on the particular characteristics of the hardware on which it executed, and reuse of programming skills learned on one kind of computer to different computers (e.g., removing the need to learn the assembly language for each new machine).

4.6.1 Programming languages

Organizations with an inventory of source code have an interest in the current and future use of programming languages: more widely used languages are likely to have more developers capable of using it (reducing hiring costs), likely to have more extensive tool

[xv] Your author has experience of companies buying hardware without realising that applications were not included in the price.

[xvi] Software user-groups are almost as old as computers.[68]

support (than less widely used languages), and are likely to be available on a wider range of platforms.

Some people enjoy creating new language, writers of science fiction sometimes go to great length to create elaborate, and linguistically viable, languages for aliens to speak.[1581] Factors found to influence the number of human languages actively used, within a geographic region, include population size, country size, and the length of the growing season;[1347] see Github–ecosystems/009_.R.

The specification of Plankalkül, the first high-level programming language, was published in 1949;[143, 2013] the specification of a still widely used language (Fortran) was published in 1954.[872] A list of compilers[xvii] for around 25 languages appears in a book[714] published in 1959 (see Github–ecosystems/Grabbe_59.txt), and in 1963 it was noted[781] that creating a programming language had become a fashionable activity. A 1976 report[597] estimated that at least 450 general-purpose languages and dialects were currently in use within the US DoD.

Many thousands of programming languages have been created, but only a handful have become widely used. Figure 4.50 shows the number of new programming languages, per year, that have been described in a published paper.

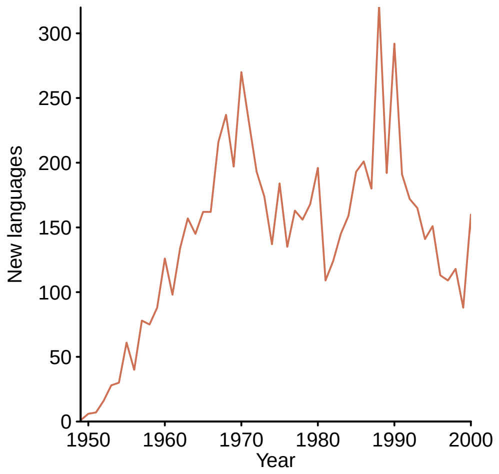


Figure 4.50: Number of new programming languages, per year, described in a published paper. Data from Pigott et al.[1467] Github–Local

Despite their advantages,[1561] high-level languages did not immediately displace machine code for the implementation of many software systems. A 1977 survey[1734] of programmers in the US Federal Government, found 45% with extensive work experience, and 35% with moderate work experience, of machine code. Developing software using early compilers could be time-consuming and labor-intensive; memory limits required compiling to be split into a sequence of passes over various representations of the source (often half-a-dozen or more[251]), with the intermediate representations being output on paper-tape, which was read back in as the input to the next pass (after reading the next compiler pass from the next paper-tape in the sequence), eventually generating assembler. Some compilers required that the computer have an attached drum-unit[105] (an early form of hard-disk), which was used to store the intermediate forms (and increase sale of peripherals). Developer belief in their own ability to produce more efficient code than a compiler was also a factor.[104]

The creation of a major new customer facing ecosystem provides an opportunity for new languages and tools to become widely used within the development community working in that ecosystem. For instance, a new language might provide functionality often required by applications targeting users of particular ecosystem, that is more convenient to use (compared to preexisting languages), alternatively there may only be one language initially available for use, e.g., Solidity for writing Ethereum contracts.

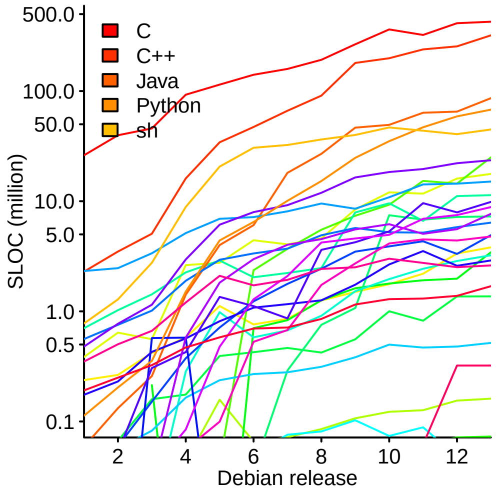


Figure 4.51: Lines of code written in the 32 programming languages appearing in the source code of the 13 major Debian releases between 1998 and 2019. Data from the Debsources developers.[457] Github–Local

Within geographical, or linguistic, separated regions the popularity of existing languages has sometimes evolved along different paths, e.g., Pascal remained popular in Russia[1181] longer than elsewhere, and in Japan Cobol remains widely used.[26]

Existing languages and their implementations evolve. Those responsible for maintaining the language specification add new constructs (which are claimed to be needed by developers), and compiler vendors need to add new features to have something new to sell to existing customers. One consequence of language evolution is that some implementations may not support all the constructs contained in the source code used to build a program.[174]

A history of the evolution[1747] of Lisp lists the forces driving its evolution: “Overall, the evolution of Lisp has been guided more by institutional rivalry, one-upsmanship, and the glee born of technical cleverness that is characteristic of the "hacker culture" than by sober assessments of technical requirements.”

There are incentives for vendors to invest in a language they control, when they also control a platform supporting an ecosystem of applications written by third-parties (e.g., C$^\sharp$ on Microsoft Windows, Objective-C on iOS, and Kotlin on Android). Writing an application in the vendor’s ecosystem language is a signal of commitment (because it entails a high switching cost).

The legal case between Oracle and Google, over Java copyright issues,[42] motivated the need for a Java compatible language that was not Java; Kotlin became available in 2017. Developers have started to use Kotlin specific features in their existing Java source.[1205]

The term *language popularity* suggests that the users of the language have some influence on the selection process, and like the language in some way. In practice developers may

[xvii] The term compiler was not always applied in the sense that is used today.

not have any say in the choice of language, and may have no positive (or negative) feelings towards the language. However, this term is in common use, and there is nothing to be gained by using something technically correct.

Figure 4.51 shows the number of lines contained in the source code of the 32 major releases of Debian, broken down by programming language (the legend lists the top five languages).

Sources of information on language use include:

- Job adverts: The number of organizations that appear to be willing to pay money to someone to use a language, is both a measure of actual usage and perceived popularity. Languages listed in job adverts are chosen for a variety of reasons including: appearing trendy in order to attract young developers to an interview, generating a smokescreen to confuse competitors, and knowledge of the language is required for the job advertised.

  A study by Davis and de la Parra[438] investigated the monthly flow of jobs on an online job market-place owned by DHI (approximately 221 thousand vancies posted, and 1.1 million applications). Figure 4.52 shows the labour market slack (calculated, for each month and keyword, as applications for all applicable vacancies divided by the sum of the number of days each vacancy was listed) for jobs postings whose description included a particular keyword.

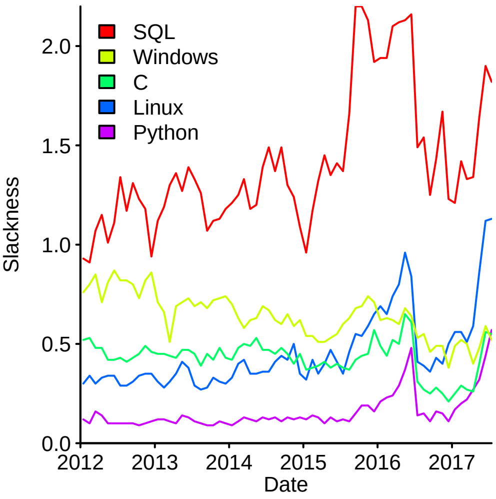


Figure 4.52: Monthly labor market slack (i.e., applications per days vacancy listed) for jobs whose description included a particular keyword (see legend). Data from Davis et al.[438] Github–Local

  Social media includes postings to employment websites and adverts for jobs. Figure 4.53 shows the number of monthly developer job related tweets that included a language name,

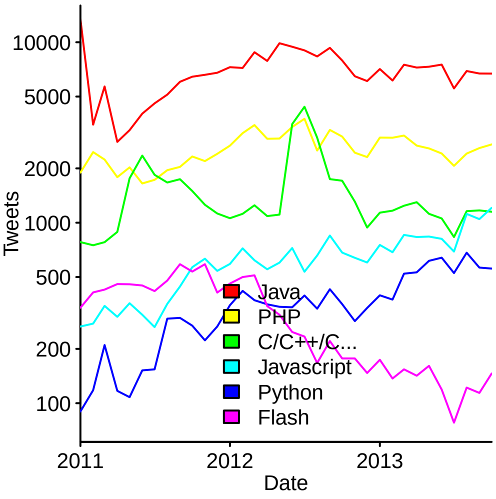


Figure 4.53: Number of monthly developer job related tweets specifying a given language. Data kindly provided by Destefanis.[479] Github–Local

- quantity of existing code: the total quantity of existing code might be used as a proxy for the number of developers who have worked with a language in the past (see fig 11.10); recent changes in the quantity of existing code is likely to provide a more up to date picture,

- number of books sold:[797] spending money on a book is an expression of intent to learn the language. The intent may be a proxy for existing code (i.e., learning the language to get a job, or work on a particular project), existing course curriculum decisions, or because the language has become fashionable.

  Sales data from individual publishers is likely to be affected by the popularity of their published books, and those of competing publishers; see Github–ecosystems/LangBooksSold.R,

- miscellaneous: prior to the growth of open source, the number of commercially available compilers was an indicator of size of the professional developer market for the language.

  Question & answer websites are unreliable indicators because languages vary in complexity, and questions tail off as answers on common issues become available. Figure 4.54 shows the normalised percentage of language related tags associated with questions appearing on Stack Overflow each month.

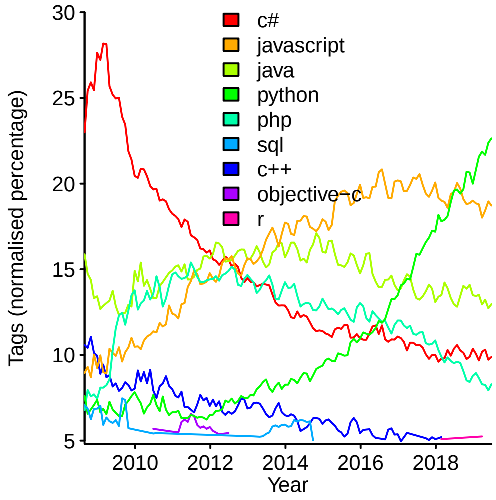


Figure 4.54: Normalised percentage of 34 language tags associated with questions appearing on Stack Overflow in each month. Data extracted from Stack Overflow website.[926] Github–Local

  Application domain specific usage, such as mobile phone development,[1541] embedded systems in general,[1818, 1840] and Docker containers.[356]

  US Federal government surveys of its own usage: a 1981 survey[383] found most programs were written in Cobol, Fortran, Assembler, Algol and PL/1, a 1995 survey[840] of 148 million LOC in DOD weapon systems Ada represented 33%, the largest percentage of any language (C usage was 22%).

  Books are a source of programming language related discussion going back many decades Language names such as Fortran and Cobol are unlikely to be used in non-programming contexts, while names such as Java and Python are more likely to be used in a non-programming context. Single letter names, such as C, or names followed by non-alphabetic characters have a variety of possible interpretations, e.g., the phrase *in C* appears in music books as a key signature, also the OCR process sometimes inserts spaces that were probably not in the original. The lack of context means that any analysis based on the English unigrams and bigrams from the Google books project[1259] is likely to be very noisey.

Applications can only be ported to a platform when compilers for the languages used are available. The availability of operating systems and device drivers written in C, means that a C compiler is one of the first developer tools created for a new processor.

Will today's widely used languages continue to be reasonably as popular over the next decade (say)? Some of the factors involved include:

- implementing a software system is expensive; it is often cheaper to continue working with what exists. The quantity of existing source contained in commercially supported applications is likely to be a good indicator of continuing demand for developers capable of using the implementation language(s),
- in a rapidly changing environment developers want to remain attractive to employers, there is a desire to have a CV that lists experience in employable languages. Perception (i.e., not reality) within developer ecosystems about which languages are considered worthwhile knowing, becoming popular or declining in usage/popularity,[1255]
- Open source developers may choose to use a language because it is the one they were first taught. Reid[1550] investigated the first language used in teaching computer science majors; between 1992 and 1999 over 20 lists of languages taught were published, with an average of 400+ universities per list. More recent surveys[1681] have been sporadic. Figure 4.55 shows the percentage of languages taught by at least 3% of the responding universities. The main language taught in universities during the 1990s (i.e., Pascal) ceased being widely used in industry during the late 1980s.

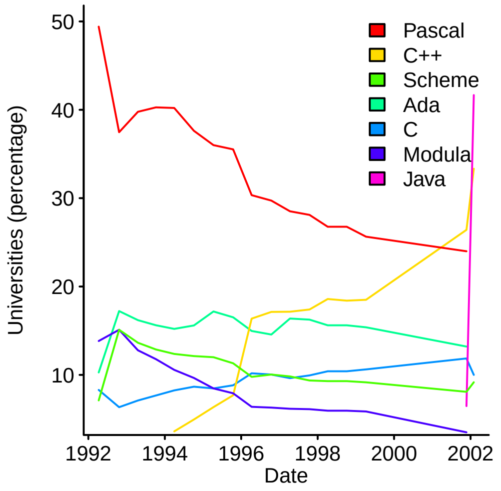


Figure 4.55: Percentage of universities reporting the first language used to teach computer science majors. Data from Reid, via the Wayback Machine,.[1550] Github–Local

Compiler vendors enhance their products by supporting features not specified in the language standard (if one exists). The purpose of adding these features is to attract customers (by responding to demand), and over time to make it more difficult for customers to switch to a different vendor. Some language implementations become influential because of the widespread use of the source code they compile (or interpret). The language extensions supported by an influential implementation have to be implemented by any implementation looking to be competitive for non-trivial use.

A study by Rigger, Marr, Adams and Mössenböck[1563] investigated the use of compiler specific built-in functions supported by gcc, in 1,842 Github projects. In C and C++ source code, use of built-ins appear as function calls, but are subject to special processing by the compiler. Analysis of the C compiler source found 12,339 distinct built-ins (some were internal implementations of other built-ins), and 4,142 unique built-ins were encountered in the C projects analysed. Figure 4.56 shows the cumulative growth in the number of Github projects supported, as support is added for more built-ins (the growth algorithm assumes an implementation order based on the frequency of unique built-in usage across projects).

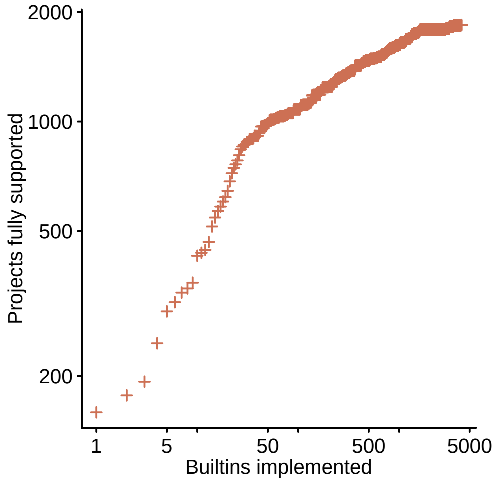


Figure 4.56: Cumulative number of Github projects that can be built as more gcc built-ins are implemented. Data from Rigger et al.[1563] Github–Local

4.6.2 Libraries and packages

Program implementation often involves functionality that is found in many other programs, e.g., opening a file and reading from it, and writing to a display device. Developers working on the first computers had access to a library of commonly used functions,[1941] and over time a variety of motivations have driven the creation and supply of libraries.

Some programming languages were designed with specific uses in mind, and include support for application domain library functions, e.g., Fortran targeted engineering/scientific users and required implementations to provide support for trigonometric functions, Cobol targeted business users, and required support for sorting.[xviii]

Manufacturers discovered that the libraries they had bundled with the operating system[201] to attract new customers, could also be used to increase customer lock-in (by tempting developers with extensive library functionality having characteristics specific to the manufacturer, that could not be easily mapped to the libraries supplied by other vendors).

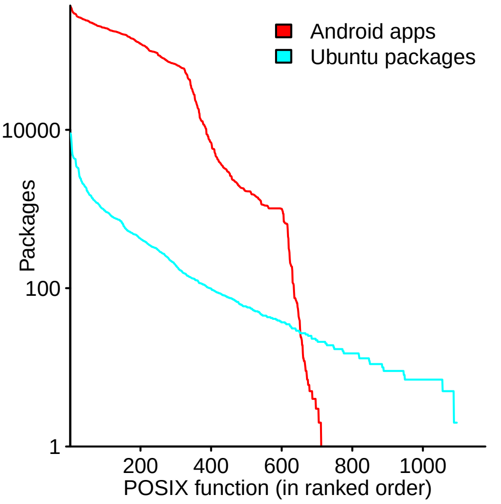


Figure 4.57: Number of Android/Ubuntu (1.1 million apps)/(71,199 packages) linking to a given POSIX function (sorted into rank order). Data from Atlidakis et al.[82] Github–Local

The decreasing cost of hardware, and the availability of an operating system, in the form of Unix source code, enabled many manufacturers to enter the minicomputer/workstation market.[296] Vendors' attempts to differentiate their product lines led to the Unix wars[1609, 1610] of the 1980s (in the mid-1990s, platforms running a Unix-derived OS typically shipped with over a thousand C/C++ header files[919]).

The POSIX standard[888] was intended provide the core functionality needed to write portable programs; this functionality was derived from existing implementation practice of widely used Unix systems.[2011] POSIX became widely supported (in part because large organizations, such as the US Government, required vendors to supply products that included POSIX support, although some implementations felt as-if they were only intended to tick a box during a tender process, rather than be used).

[xviii] The C Standard specifies support for some surprising functions, surprising until it is realised that they are needed to implement a C compiler, e.g., `strtoul`.

A study by Atlidakis, Andrus, Geambasu, Mitropoulos and Nieh[82] investigated POSIX usage (which defines 1,177 functions) across Android 4.3 (1.1 million apps measured, 790 functions tracked, out of 821 implemented) and Ubuntu 12.04 (71,199 packages measured, 1,085 functions tracked, out of 1,115 implemented). Figure 4.57 shows the use of POSIX functions by Apps/packages available for the respective OS (these numbers do not include calls to `ioctl`, whose action is to effectively perform an implementation defined call).

Linux came late to the Unix wars, and emerged as the primary base Unix kernel. The Linux Standard Base[xix] is intended to support a binary compatibility interface for application executables; this interface includes pseudo-file systems (e.g., `/proc`) that provide various kinds of system information.

A study by Tsai, Jain, Abdul and Porter[1827] investigated use of the Linux API by the 30,976 packages in the Ubuntu 15.04 repository, including system calls, calls to function in `libc`, `ioctl` argument value, and pseudo-file system usage (the measurements were obtained via static analysis of program binaries). Figure 4.58 shows each API use (e.g., call to a particular function or reference to a particular system file) provided by the respective service, as a percentage of the possible 30,976 packages that could reference them, in API rank order.

The internet created a new method of software distribution: a website. People and organizations have built websites to support the distribution of packages available for a particular language (rather than a particular OS), and some have become the primary source of third-party packages for their target language, e.g., `npm` for Javascript, and `CRAN` for `R`.

Publicly available package repositories are a relatively new phenomena; a few were started in the mid-1990s (e.g., `CRAN` and `CPAN`), and major new repositories are still appearing 15-years later (e.g., Github in 2008 and `npm` in 2010). The age of a repository can have a large impact on the results of an analysis of the characteristics measured, e.g., rate of change is likely to be high in the years immediately after a repository is created.

Package repositories are subject to strong network effects. Developers want to minimise the effort invested in searching for packages, and the repository containing the most packages is likely to attract the most searches; also, the number of active users of a repository is a signal for authors of new packages, seeking to attract developers, who need to choose a distribution channel.

A study by Caneill and Zacchiroli[292] investigated package usage in the official Debian releases and updates. Figure 4.59 shows the survival curve of the latest version of a package included in 10 official Debian releases, along with the survival of the same version of a package over successive releases.

A niche market may be large enough, and have sufficiently specialist needs, for a repository catering to its requirements to become popular within the niche, e.g., the Bioconductor website aims to support the statistical analysis of biological assays and contains a unique collection of `R` packages targeting this market.

Figure 4.60 shows the number of `R` packages in three large repositories (a fourth, `R`-forge is not displayed), along with the number of packages present in multiple repositories (colored areas not scaled by number of packages). Each website benefits from its own distinct network effects, e.g., Github provides repositories, and gives users direct control of their files.

The requirements, imposed by the owners of a repository, for a package to be included, may add friction to the workflow of a package under active development (e.g., package authors may want the latest release to be rapidly available to potential users). For instance, some `R` packages under active development are available on GitHub, with updates being submitted to `CRAN` once they are stable (some package have dependencies on packages in the other repositories, and dependency conflicts exist[461]).

In some cases a strong symbiotic relationship between a language and package ecosystem has formed, which has influenced the uptake of new language revisions, e.g., one reason for the slowed uptake of Python 3 has been existing codes' use of packages that require the use of Python 2.

Many packages evolve, and versioning schemes have been created to provide information about the relative order of releases and the compatibility of a release with previous

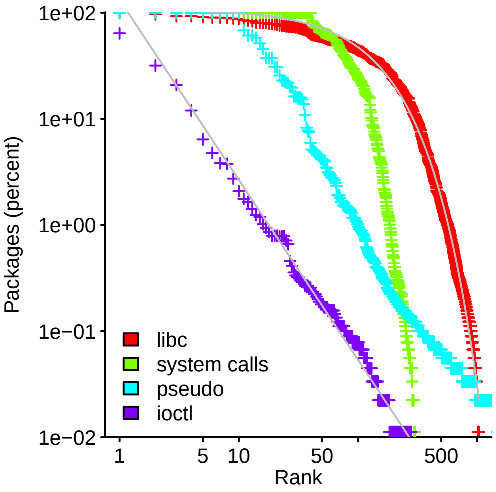


Figure 4.58: Percentage of packages referencing a particular API provided by a given service (sorted in rank order); grey lines are a fitted power law and exponential, to `ioctl` and `libc` respectively. Data from Tsai et al.[1827] Github–Local

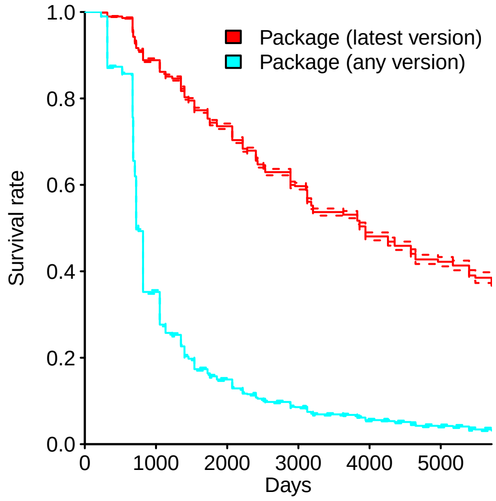


Figure 4.59: Survival curve of packages included in 10 official Debian releases, and inclusion of the same release of a package; dashed lines are 95% confidence intervals. Data from Caneill et al.[292] Github–Local

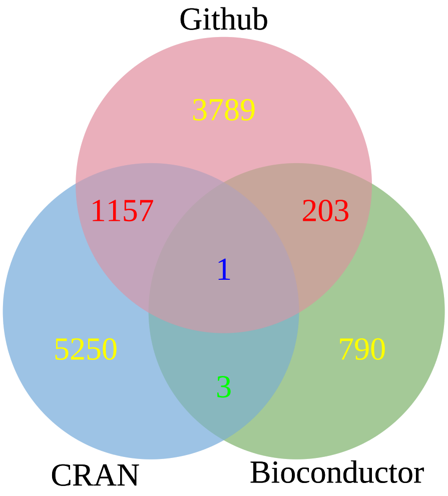


Figure 4.60: Number of packages in three widely used `R` repositories (during 2015), overlapping regions show packages appearing in multiple repositories (areas not to scale). Data from Decan et al.[460] Github–Local

[xix] LSB 3.1 was first published as an ISO Standard in 2006, it was updated to reflect later versions of the LSB in. The Linux API has evolved.[110]

releases. The semver[1510] semantic versioning specification has become widely used; the version string contains three components denoting a major release number (not guaranteed to be compatible with previous releases), minor release number (when new functionality is added), and patch number (correction of mistake(s)).

Package managers often provide a means of specifying version dependency constraints, e.g., `^1.2.3` specifies a constraint that can be satisfied by any version between `1.2.3` and `2.0.0`. Studies[458] have found that package developers' use of constraints does not always fully comply with the semver specification.

The installation of a package may fail because of dependency conflicts between the packages already installed and this new package, e.g., installed package P_1 may depend on package P_2 having a version less than 2.0, while package P_3 depends on package P_2 having a version of at least 2.0. A study by Drobisz, Mens and Di Cosmo[504] investigated Debian package dependency conflicts. Figure 4.61 shows the survival curve of package lifetime, and the interval to a package's first conflict.

A study by Decan, Mens and Claes[459] investigated how the package dependencies of three ecosystems changed over time (`npm` over 6-years, `CRAN` 18-years, and `RubyGems` over 7-years). The dependencies in roughly two-thirds of `CRAN` and `RubyGems` packages specified greater-than or equal to some version, with the other third not specifying any specific version; `npm` package dependencies specified greater-than in just under two-thirds of cases, with a variety of version strings making up the other uses (particular version numbers were common); see Github–ecosystems/SANER-2017.R

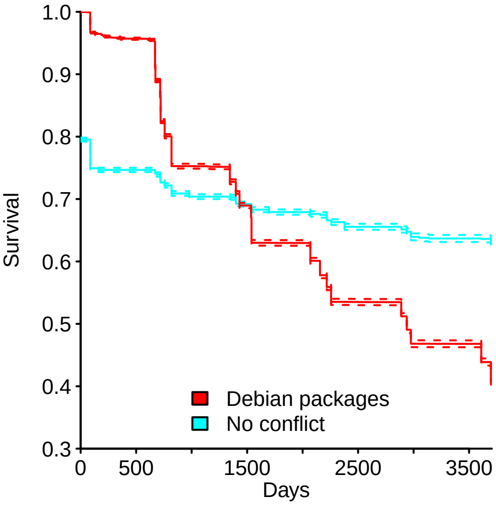


Figure 4.61: Survival curves for Debian package lifetime and interval before a package contains its first dependency conflict; dashed lines are 95% confidence intervals. Data from Drobisz et al.[504] Github–Local

Third-party libraries may contain undocumented functionality that is accessible to developers. This functionality may be intended for internal library use, and is only accessible because it is not possible to prevent developers accessing it; or, the vendor intent is to provide privileged access to internal functionality for their own applications, e.g., Microsoft Word using undocumented functionality in the libraries available in Microsoft Windows.[650]

There may be a short term benefit for developers making use of internal functionality, that makes it worth the risk of paying an unknown cost later if the internal functionality is removed or modified; see fig 11.76. Information on undocumented functionality in Microsoft Windows was widely available,[1633] with major Windows updates occurring every three years, or so.

Major updates to Android have occurred once or twice a year (see fig 8.37), and figure 4.62 shows how the lifetime of internal functionality has been declining exponentially.

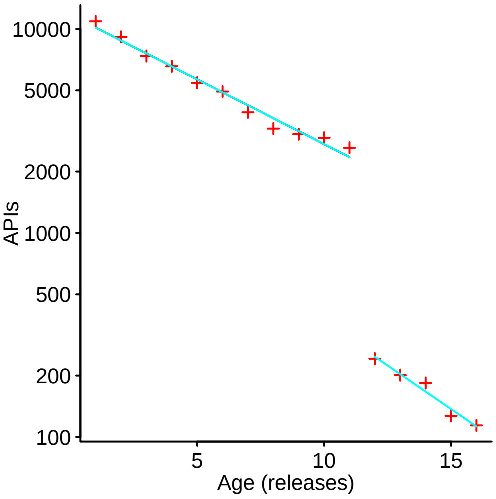


Figure 4.62: Number of Android APIs surviving a given number of releases (measured over 17 releases), with fitted regression lines. Data from Li et al.[1114] Github–Local

4.6.3 Tools

Computer manufacturers were the primary suppliers of software development tools until suppliers of third-party tools gained cost-effective access to the necessary hardware, and the potential customer base became large enough to make it commercially attractive for third-parties to create and sell tools.

When developer tools are sold for profit, the vendor has an incentive to keep both customer and the tool users happy.[xx] Open source has significantly reduced the number of customers for some tools (e.g., compilers), without decreasing the number of users. A company creating their own cpu needs to support at least a C compiler. Funding the implementation of a code generator, based on an Open source compiler allows them to make available a free compiler tool chain for their cpu; the users of this tool chain are the actual product.

The development environment in which tools are used can have a big impact on the functionality provided, with sophisticated functionality being dropped in new products only to reappear again many years later. For instance, during the early years of computing, interaction with computers was usually via batch processing, rather than one-line access;[686] users submitted a job for the computer to perform (i.e., a sequence of commands to execute), and it joined the queue of jobs waiting to be run. Developers might only complete one or two edit/compile/execute cycles per day. In this environment, high quality compiler error recovery can significantly boost developer productivity; having an executable for an

[xx] Customers pay money, users use; a developer can be both a customer and a user.

error corrected version of the submitted source (along with a list of errors found), gives developers a binary that may be patched to do something useful.[xxi] Borland's Turbo-Pascal stopped compiling at the first error it encountered, dropping the user into an editor with the cursor located at the point the error was detected. Compilation was so fast, within its interactive environment on a personal computer, developers loved using it. Error recovery in modern compilers, at the time of writing, has yet to return to the level of sophistication available in some early mainframe compilers.

Adding support for new language features, new processors and environments is a source of income for compiler vendors; the performance of code generated by compilers are often benchmarked, to find which generates the faster and/or more compact code.

Compile time options provide a means for users to tailor compiler behavior to their needs. Figure 4.63 shows the number of options supported by gcc for specific C and C++ options, and various compiler phases, for 96 releases between 1999 and 2019; during this time the total number of supported options grew from 632 to 2,477.

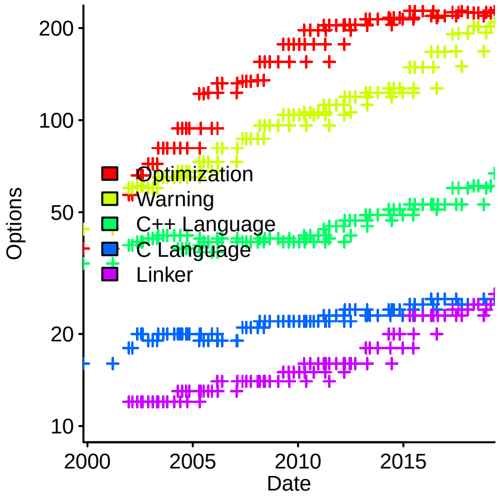


Figure 4.63: Number of gcc compiler options, for all supported versions, relating to languages and the process of building an executable program. Data extracted from gcc website.[654] Github–Local

Support for an option is sometimes removed; of the 1,091 non-processor specific options supported, 214 were removed before release 9.1. Since 1999, the official release of GCC has included support for 80 cpus, with support for 20 of these removed before release 9.1 (see fig 4.13).

4.6.4 Information sources

Readily available sources of reliable information can help reduce the friction of working within an ecosystem, and can significantly reduce the investment that has to be made by outsiders to join an ecosystem.

Vendors that support an API have to document the available interfaces and provide examples, if they want developers, third-party or otherwise, to make use of the services they provide. APIs evolve, and the documentation has to be kept up todate.[1673] The growth of popular APIs, over time, can result in an enormous collection of documentation, e.g., the 130+ documents for the Microsoft Server protocol specifications[1260] contain over 16 thousand pages (see fig 8.24). Figure 4.64 shows how the number of words in Intel's x86 architecture manual has grown over time.

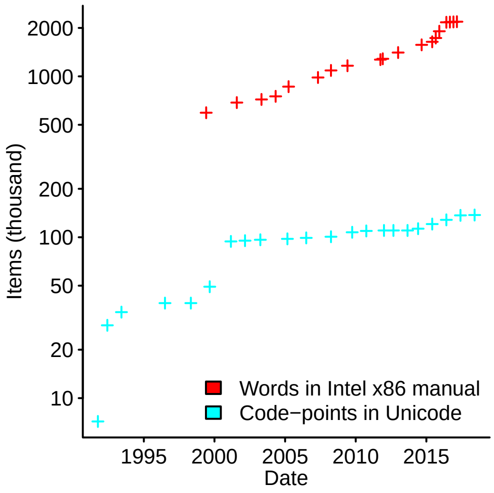


Figure 4.64: Words in Intel x86 architecture manuals, and code-points in Unicode Standard over time. Data for Intel x86 manual kindly provided by Baumann.[145] Github–Local

Books are a source of organized and collated material. Technical books are rarely profitable for their authors, but can act as an advert for the author's expertise, who may offer consultancy or training services. A high quality book, or manual, may reduce the size of the pool of potential clients, but is a signal to those remaining of a potential knowledgeable supplier of consultancy/training.

While executable source code is definitive, comments contained in code may be incorrect or out of date; in particular links in source code may cease to work.[776]

[xxi] Your author once worked on a compiler project, funded with the aim of generating code 60% smaller than the current compiler. Developers hated this new compiler because it generated very little redundant code; the redundant code generated by the previous compiler was useful because it could be used to hold patches.

Chapter 5

Projects

5.1 Introduction

The hardest part of any project is, generally, obtaining the funding to implement it.[1636]

Clients[i] are paying for a solution to a problem, and have decided that a software system is likely to provide a cost effective solution. The client might be a large organization contracting a third party to develop a new system, or update an existing system, a company writing software for internal use, or an individual spending their own time scratching an itch (who might then attract other developers[801]).

Successfully implementing a software system involves creating a financially viable implementation that solves the client's problem. Financial viability means not so expensive the client is unable to pay for it, and not so cheap that the software vendor fails to make a reasonable profit.

Commercial software projects aim to implement the clients' understanding of the world (in which they operate, or want to operate), in a computer executable model that can be integrated into the business, and be economically operated.

It can be unwise to ask clients why they want the software. Be thankful that somebody is willing to pay to have bespoke software written,[746] creating employment for software developers. For instance:

"The first go-around at it was about $750 million, so you figure that's not a bad cost overrun for an IT project. Then I said, "Well, now, tell me. Did you do an NPV? Did you do a ROI? What did you do on a $750 million IT investment?" And she sort of looked a little chagrined and she said, "Well, actually, there was no analysis done on that." I said, "Excuse me . . . can you explain that to me please. That's not what the textbook says." She said, "Well, it was a sales organization, the brokers worked for the sales organization." The sales organization — this was a few years ago when the brokerage business was extremely good — said, "you know, the last two years we've made more than enough money to pay for this. We want it, and we're going to pay for it." And the board of directors looked at how much money they were making and they said, "You go pay for it". So that was the investment analysis for a $750 million IT investment that turned into a billion dollars."[1788]

The difference between projects in evidence-based engineering disciplines (e.g., bridge building in civil engineering), and projects in disciplines where evidence-based practices have not been established (e.g., software development), is that in the former implementation involves making the optimum use of known alternatives, while the latter involves discovering what the workable alternatives might be (e.g., learning, by trying ideas until something that works well enough is discovered).

Building a software system is a creative endeavour; however, it differs from artistic creative endeavours in that the externally visible narrative has to interface with customer reality. Factory production builds duplicates based on an exemplar product, and can be a costly process; software duplication is a trivial process that has virtually zero cost.

Useful programs vary in size from a few lines of code, to millions of lines, and might be written by an individual in under an hour, or by thousands of developers over many years.

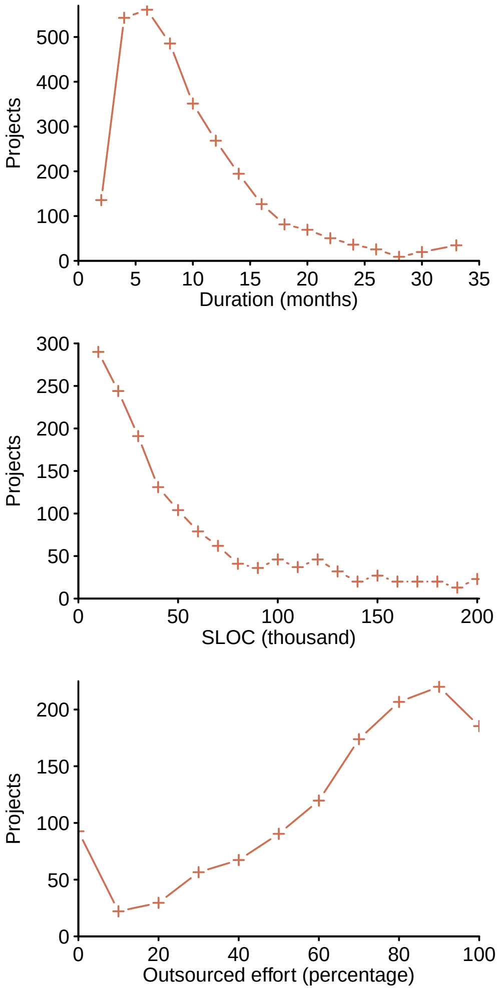


Figure 5.1: Number of projects having a given duration (upper; 2,992 projects), delivered containing a given number of SLOC (middle; 1,859 projects), and using a given percentage of out-sourced effort (lower; 1,267 projects). Data extracted from Akita et al.[26] Github–Local

[i] The term *customer* has mass market associations, bankrolling bespoke software development deserves something having upmarket connotations.

Much of the existing software engineering research has focused on large projects, reasons for this include: the bureaucracy needed to support large projects creates paperwork from which data can be extracted for analysis, and the large organizations providing the large sums needed to finance large projects are able to sway research agendas. This book is driven by the availability of data, and much of this data comes from large projects and open source; small commercial projects are important; they are much more common than large projects, and it is possible they are economically more important.

What are the characteristics of the majority of software projects? Figure 5.1, using data from a multi-year data collection process[26] by the Software Engineering Center, of Japan's Information-Technology Promotion Agency, shows that most are completed in under a year, contain less than 40 KSLOC and that much of the effort is performed by external contractors

When approached by a potential client, about creating a bespoke software system, the first question a vendor tries to answer is: does the client have the resources needed to pay for implementing such a software system? If the client appears to be willing to commit the money (and internal resources), the vendor may consider it worth investing to obtain a good-enough view on the extent to which the client desires can be implemented well enough for them to pay for a bespoke software system to be built.

The head of NASA told President Kennedy that $20 billion was the price tag for landing on the Moon (NASA engineers had previously estimated a cost of $10-12 billion[1645]); the final cost was $24 billion.

A study by Anda, Sjøberg and Mockus[52] sent a request to tender to 81 software consultancy companies in Norway, 46 did not respond with bids. Figure 5.2 shows the estimated schedule days, and bid price received, from 14 companies (21 companies provided a bid price without a schedule estimate). Fitting a regression model that includes information on an assessment of the analysis, and design performed by each company, along with estimated planned effort, explains almost 90% of the variation in the schedule estimates; see Github–projects/effort-bidprice.R.

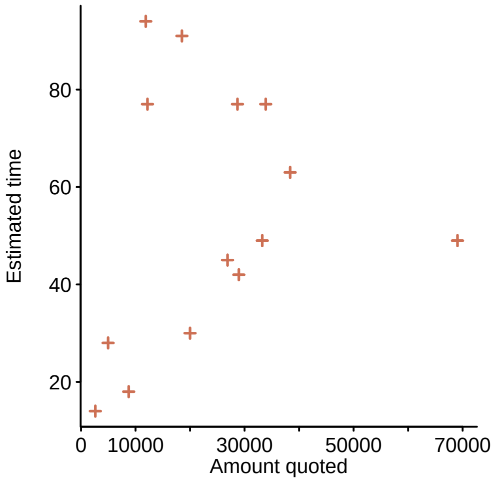


Figure 5.2: Firm bid price (in euros) against schedule estimate (in days), received from 14 companies, for the same tender specification. Data from Anda et al.[52] Github–Local

Both the client and the vendor want their project to be a success. What makes a project a success?

From the vendor perspective (this book's point of view), a successful project is one that produces an acceptable profit[ii]. A project may be a success from the vendor's perspective, and be regarded as a failure by the client (because the client paid, but did not use the completed system,[21] or because users under-utilised the delivered system, or avoided using it[1084]), or by the developers who worked on the project (because they created a workable system despite scheduling and costing underestimates by management[1138]). These different points of view mean that the answers given to project success surveys[49] are open to multiple interpretations.

A study by Milis[1264] investigated views of project success by professionals in the roles of management, team member (either no subsequent project involvement after handover, or likely to receive project benefits after returning to their department), and end-user. Fitting regression models to the data from 25 projects, for each role, finds one explanatory variable of project success common to all roles: user happiness with the system. Management considered being within budget as an indicator of success, while other roles were more interested in meeting the specification; see Github–projects/Milis-PhD.R for details.

A project may fail because the users of a system resist its introduction into their work-flow,[1073] e.g., they perceive its use as a threat to their authority. Failure to deliver a system can result in the vendor having to refund any money paid by the client, and make a payment for work performed by the client.[83]

The Queensland Health Payroll System Commission of inquiry report[340] provides a detailed analysis of a large failed project.

A study by Zwikael and Globerson[2014] investigated project cost and schedule overruns, and success in various industries, including software. The results suggest that except for the construction industry, overrun rates are broadly similar.

Almost as soon as computers became available million line programs were being written to order. Figure 5.3 shows lines of code and development costs for US Air Force software projects, by year, excluding classified projects and embedded systems; the spiky nature of the data suggests that LOC and development costs are counted in the year a project is delivered.

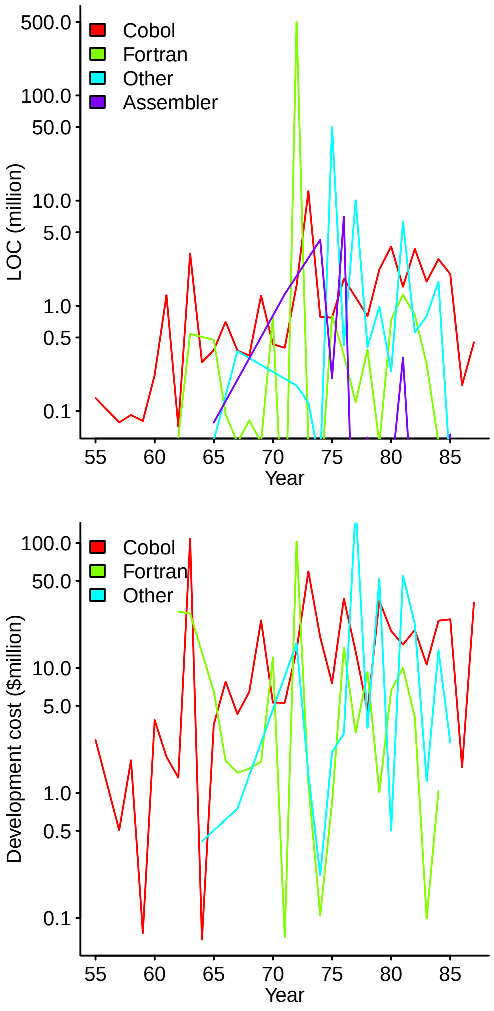


Figure 5.3: Annual development cost and lines of code delivered to the US Air Force between 1960 and 1986. Data extracted from NeSmith.[1346] Github–Local

[ii] Research papers often use keeping within budget and schedule as measures of success; this begs the question of which budget, and which schedule, e.g., the initial, final, intermediate, or final versions?

5.1.1 Project culture

Software projects are often implemented within existing business cultural frameworks, which provides the structure for the power and authority to those involved. A project's assigned or perceived power and authority controls the extent to which either: outsiders have to adapt to the needs of the project, or the project has to be adapted to the ecosystems in which it is being created and is intended to be used. The following are some common development culture frameworks:

- one-off projects within an organization, which treats software as a necessary investment that has to be made to keep some aspect of the business functioning. Employees involved in the development might treat working on the project as a temporary secondment that is part of what they perceive to be their main job, or perhaps a stepping stone to a promotion, or a chance to get hands-on experience developing software (maybe with a view to doing this kind of job full time). External contractors may be hired to work on the project,
- projects within an organization, which derives most of its income from software products, e.g., software developed to be sold as a product (see fig 5.56). In the case of startups, the projects implemented early in the company's life may be produced in an organizational vacuum, and thus have the opportunity to mold the company culture experienced by future projects,
- projects within an organization, where software is the enabler of the products and services that form the basis of its business model, e.g., embedded systems.

  A study by Powell[1497] investigated software projects for engine control systems at Rolls-Royce. Figure 5.4 shows effort distribution (in person hours) over four projects (various colors), plus non-project work (blue) and holidays (purple'ish, at the top), over 20 months. Staff turnover and use of overtime during critical periods means that total effort changes over time (also see fig 11.73).

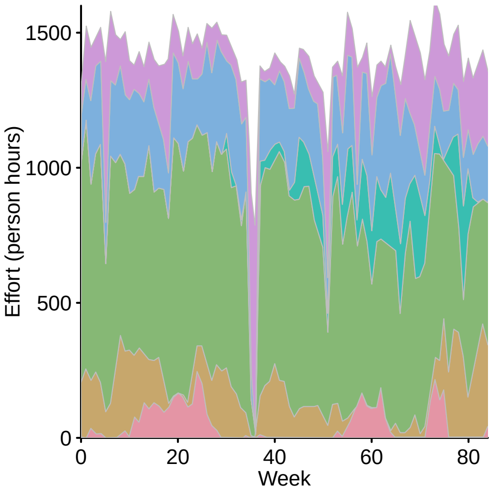


Figure 5.4: Distribution of effort (person hours) during the development of four engine control system projects (various colors), plus non-project work (blue) and holidays (purple'ish, at top), at Rolls-Royce. Data extracted from Powell.[1497] Github–Local

  In its 2015 financial year Mozilla, a non-profit company that develops and supports the Firefox browser, had income of $421,275 million and spent $214,187 million on software development.[1770] Almost all of this income came from Google, who paid to be Firefox's default search engine,

- contract software development, where one-off projects are paid for by other organization for their own use. Companies in the contract software development business need to keep their employees busy with fee earning work, and assigning them to work on multiple projects is one way of reducing the likelihood of an employee not producing income, i.e., while a single project may be put on hold until some event occurs, multiple projects are less likely to be on hold at the same time.

  Companies in the business of bespoke software development have to make a profit on average, over all projects undertaken within some period, i.e., they have some flexibility in over- and underestimating the cost of individual projects. Figure 5.5 shows the percentage loss/profit made by a software house on 146 fixed-price projects.

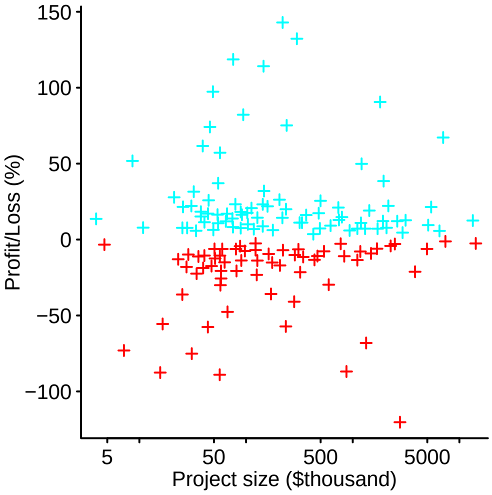


Figure 5.5: Percentage profit/loss on 145 fixed-price software development contracts. Data extracted from Coombs.[391] Github–Local

- projects where the participants receive income in the form of enjoyment derived from the work involved; the implementation is part of the developers' lifestyle.

  The characteristics of single person projects are likely to experience greater variation than larger projects, not because the creator is driven by hedonism, but because a fixed release criteria may not exist, and other activities may cause work to be interrupted for indefinite periods of time (i.e., measurements of an individual is likely to have greater variance than a collection of people).

  A few single developer projects grow to include hundreds of paid and volunteer developers. The development work for projects such as Linux[394] and GCC is spread over multiple organizations, making it difficult to estimate the level of commercial funding.

  Code commits made by volunteer developers are less likely to occur during the working week than commits made by paid developers. Figure 5.6 shows the hourly commits during a week (summed over all commits) for Linux and FreeBSD. The significant difference in number of commits during the week, compared to the weekend, suggests that Linux has a higher percentage of work performed by paid developers than FreeBSD.

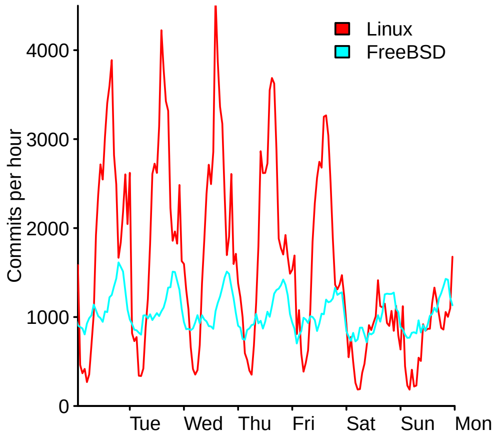


Figure 5.6: Commits within a particular hour and day of week for Linux and FreeBSD. Data from Eyolfson et al.[558] Github–Local

Academic software projects can appear within any of these categories (with personal reputation being the income sought[1312]).

Those involved in software projects, like all other human activities, sometimes engage in subversion and lying,[1586] activities range from departmental infighting to individual rivalry, and motivations include: egotism, defending one's position, revenge, or a disgruntled employee.

5.1.2 Project lifespan

Projects continue to live for as long as they are funded, and can only exceed their original budget when the client is willing to pay (because they have an expectation of delivery). Having a solution to some problems is sufficiently important that clients have no choice but to pay what it takes, and wait for delivery.[1234] For instance, when not having a working software system is likely to result in the client ceasing to be competitive, resulting in ruin or a loss significantly greater than the cost of paying for the software.

The funding of some projects is driven by company politics,[178] and/or senior management ego, and a project might be cancelled for reasons associated with how it came to be funded, independently of any issues associated with project performance.

A non-trivial percentage of projects, software and non-software,[1233] are cancelled without ever being used, e.g., VMware cancelled the project[270] to enhance their x86 emulator to support the x86 architecture functionality needed to support OS/2. Open source and personal projects are not cancelled as-such, those involved simply stop working on them.[369] The cost-effectiveness of any investment decision (e.g., investing to create software that is cheaper to maintain) has to include the risk that the project is cancelled.

- a study by El Emam and Koru[530] surveyed 84 midlevel and senior project managers, between 2005 and 2007; they found the majority reporting IT project cancellation rates in the range 11-40%. A study by Whitfield[1929] investigated 105 outsourced UK government related ICT (Information and Communication Technology) projects between 1997 and 2007 having a total value of £29.6 billion; 57% of contracts experienced cost overruns (totalling £9.0 billion, with an average cost overrun of 30.5%), of which 30% were terminated,
- the 1994 CHAOS report[1736] is a commonly cited survey of project cost overruns and cancellations. This survey is not representative because it explicitly focuses on failures, subjects were asked: " . . . to share failure stories.", i.e., the report lists many failures and high failure rates because subjects were asked to provide this very information. The accuracy of the analysis used to calculate the summary statistics listed in the report has been questioned.[557,944]

A study by McManus and Wood-Harper[1229] investigated the sources of failure of information systems projects. Figure 5.7 shows the survival curve for 214 projects, by development stage.

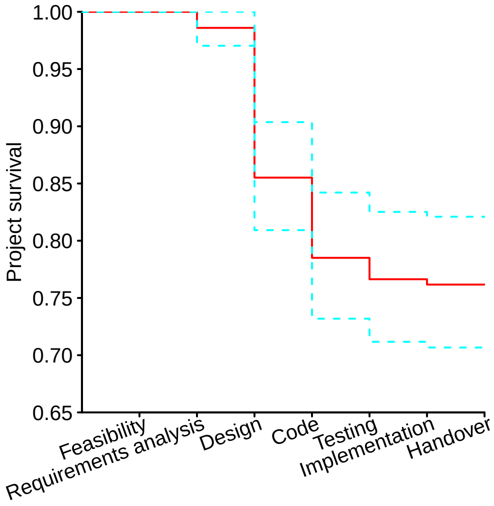


Figure 5.7: Survival rate of 214 projects, by development stage, with 95% confidence intervals. Data from McManus et al.[1229] Github–Local

Software systems are sometimes used for many years after development on them has stopped.

5.2 Pitching for projects

Bidding on a request for tender, convincing senior management to fund a project, selling the benefits of a bespoke solution to potential a client: all involve making commitments that can directly impact project implementation. An appreciation of some of the choices made when pitching for a project is useful for understanding why things are the way they are.

What motivates anyone to invest the resources needed to form a good enough estimate to implement some software?

The motivation for bidding on a tender comes from the profit derived from winning the implementation contract, while for internal projects, developers are performing one of the jobs they are employed to perform.

A study by Moløkken-Østvold, Jørgensen, Tanilkan, Gallis, Lien and Hove[1290] investigated 44 software project estimates made by 18 companies; the estimates were either for external client projects, or internal projects. A fitted regression model finds that estimated project duration was longer for internal projects, compared to external client projects; see Github–projects/RK31-surveycostestim.R.

Figure 5.8 shows the estimated and actual effort for internal (red) and external (blue) projects, along with fitted regression models. Most estimates are underestimates (i.e., above the green line); for smaller projects external projects are more accurately estimated than internal estimates, but for larger projects internal projects are more accurately estimated.

A client interested in funding the development of bespoke software may change their mind, if they hear a price that is much higher than expected, or a delivery date much later than desired. Optimal frog boiling entails starting low, and increasing at a rate that does not disturb. However, estimates have to be believable (e.g., what do clients consider to be the minimum credible cost, and what is their maximum willing to spend limit), and excessive accuracy can cause people to question the expertise of those providing the estimate.[1149]

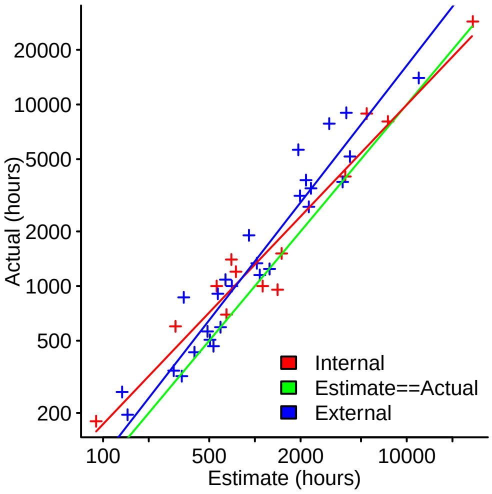


Figure 5.8: Estimated and Actual effort for internal and external projects, lines are fitted regression models; both lines are fitted regression models of the form: *Actual* ∝ *Estimate*^{a}, where a takes the value 0.9 or 1.1. Data from Moløkken-Østvold et al.[1290] Github–Local

Companies in the business of developing bespoke software need to maintain a pipeline of projects, opening with client qualification and closing with contract signature.[1711] Acquiring paying project work is the responsibility of the sales department, and is outside the scope of this book.

Many of the factors involved in project bidding are likely to be common to engineering projects in general, e.g., highway procurement.[115] Bidding decisions can be driven by factors that have little or no connection with the technical aspects of software implementation, or its costs; some of these factors include:

- likelihood of being successful on bids for other projects. If work is currently scarce, it may be worthwhile accepting a small profit margin, or even none at all, simply to have work that keeps the business ticking over. A software development organization, whether it employs one person or thousands, needs to schedule its activities to keep everyone busy, and the money flowing in.

  Some of the schedule estimates in figure 5.2 might be explained by companies assigning developers to more than one project at the same time; maintaining staff workload by having something else for them to do, if they are blocked from working on their primary project,
- bid the maximum price the client is currently willing to pay; a profitable strategy when the client estimate is significantly greater than the actual. If the client perceives a project to be important enough, they are likely to be willing to pay more once existing monies have been spent. From the client perspective, it is better to pay £2 million for a system that provides a good return on investment, than the £1 million actually budgeted, if the lower price system is not worth having,
- the likely value of estimates submitted by other bidders; competition is a hard taskmaster,
- bidding low to win, and ensuring that wording in the contract allows for increases due to unanticipated work. Prior experience shows that clients often want to change the requirements, and estimates for these new requirements are made after the competitive bidding process (see fig 3.23). The report by the Queensland health payroll system commission of inquiry[340] offers some insight into this approach. Clients often prefer to continue to work with a supplier who has run into difficulties,[227] even substantial ones,
- bidding low to win, with the expectation of recouping any losses and making profit during the maintenance phase. This strategy is based on having a reasonable expectation that the client will use the software for many years, that the software will need substantial maintenance, and that the complexity of the system and quality of internal documentation will deter competitive maintenance bids,
- bidding on projects that are small, relative to the size of the development company, as a means of getting a foot in the door to gain access to work involving larger projects, e.g., becoming an approved supplier.

The bidding process for projects usually evolves over time.

A study by Jørgensen and Carelius investigated the impact of changes to the project specification on the amount bid. Estimators in group A made an estimate based on a one-page specification, and sometime later a second estimate based on an eleven-page specification; estimators in group B made an estimate based on the eleven-page specification only. Figure 5.9 shows bids made by two groups of estimators from the same company; for additional analysis, see Github–projects/proj-bidding.R.

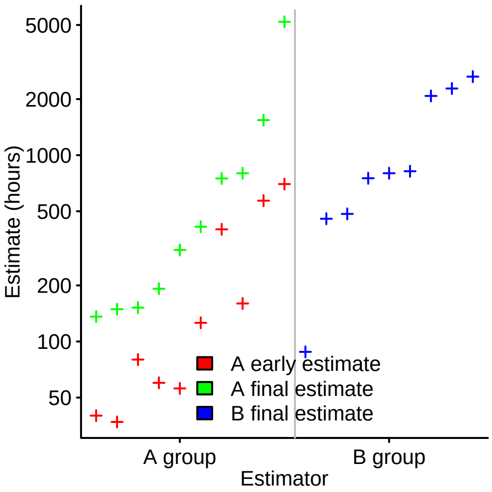


Figure 5.9: Bids made by 19 estimators from the same company (divided by grey line into the two experimental groups). Data from Jørgensen et al.[938] Github–Local

The signing of a contract signals the start of development work, not the end of client cost negotiation.

The bidding on a project for a new IT system for Magistrates' Courts (the Libra project),[227] started with ICL submitting a bid of £146 million, when it became public there was only one bidder this was increased to £184 million over 10.5 years, a contract was signed, then a revised contract was renegotiated for £319 million, then ICL threatened to repudiate the renegotiated contract and proposed a new price of £400 million, then reduced its proposed price to £384 million, and after agreement could not be reached signed a revised contract for £232 million over 8.5 years.

5.2.1 Contracts

Contract signing is the starting point for investing resources in project implementation (although some projects never involve a contract, and some work may start before a contract is signed). Having to read a contract after it has been signed is an indication that one of the parties involved is not happy; nobody wants to involve lawyers in sorting out a signed contract.[1217]

A practical contract includes provisions for foreseeable items such as client changes to the requirements and schedule slippage. Developers and clients can have very different views about the risks involved in a project.[1048]

What is the payment schedule for the project? The two primary contract payment schedules are *fixed price* and *time and materials* (also known as *cost plus*; where the client pays the vendors costs plus an agreed percentage profit, e.g., a margin of 10-15%[381]).

On projects of any size, agreed payments are made at specified milestones. Milestones are a way for the client to monitor progress, and the vendor to receive income for the work they have done. The use of milestones favours a sequential development viewpoint, and one study[81] found that the waterfall model is effectively written into contracts.

From the client's perspective a fixed price contract appears attractive, but from the vendor's perspective this type of contract may be unacceptably risky. One study[701] found that vendors preferred a fixed price contract when they could increase their profit margin by leveraging particular staff expertise; another study[1123] found that fixed-price was only used by clients for trusted vendors. Writing a sufficiently exact specification of what a software system is expected to do, along with tightly defined acceptance criteria is time-consuming and costly, which means the contract has to contain mechanisms to handle changes to the requirements; such mechanisms are open to exploitation by both clients and vendors.

A time and materials contract has the advantage that vendors are willing to accept open-ended requirements, but has the disadvantage (to the client) that the vendor has no incentive to keep costs down.

Contracts sometimes include penalty clauses and incentive fees (which are meaningless unless linked to performance targets[539]).

A study by Webster[1918] analysed legal disputes involving system failures in the period 1976-2000 (120 were found). The cases could be broadly divided into seven categories: client claims the installed system is defective in some way and vendor fails to repair it, installed system does not live up to the claims made by the vendor, a project starts and the date of final delivery continues to slip and remain in the future, unplanned obsolescence (client discovers that the system either no longer meets its needs or that the vendor will no longer support it), the vendor changes the functionality or configuration of the system resulting in unpleasant or unintended consequences for one or more clients, a three-way tangle between vendor, leasing company and client, and miscellaneous.

Many commercial transactions are governed by standard form contracts. A study by Marotta-Wurgler[1195] analysed 647 software license agreements from various markets; almost all had a net bias, relative to relevant default rules, in favor of the software company (who wrote the agreement).

A contract involving the licensing of source code, or other material, may require the licensor to assert that they have the rights needed to enter into the licensing agreement; intellectual property licensing is discussed in section 3.3.1. Project management has to be vigilant that developers working on a project do not include material licensed under an agreement that is not compatible with the license used by the project, e.g., material that has been downloaded from the Internet.[1712]

It is possible for both the client and vendor to be in an asymmetric information situation, in terms of knowledge of the problem being solved, the general application domain, and what software systems are capable of doing; the issue of moral hazard is discussed in section 3.4.7.

A study by Ahonen, Savolainen, Merikoski and Nevalainen[22] investigated the impact of contract type on reported project management effort for 117 projects; 61 projects had fixed-price contracts, and 56 time-and-materials contracts.

Figure 5.10 shows project effort (in thousands of hours) against percentage of reported management time, broken down by fixed-priced and time-and-material contracts; lines are fitted regression models. Fitting a regression model that includes team size (see Github–projects/ahonen2015.R), finds that time-and-materials contracts involve 25% less management time (it is not possible to estimate the impact of contract choice on project effort).

Tools intended to aid developers in checking compliance with the contractual rights and obligations specified in a wide range of licenses are starting to become available.[1184]

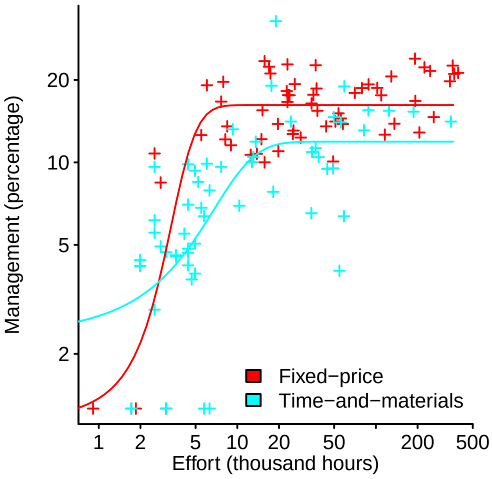


Figure 5.10: Project effort, in thousand hours, against percentage of management time, broken down by contract type; both lines are fitted logistic equations with maximums of 12% and 16%. Data extracted from Ahonen.[22] Github–Local

5.3 Resource estimation

Resource estimation is the process of calculating a good enough estimate[iii] of the resources needed to create software whose behavior is partially specified, when the work is to be done by people who have not previously written software having the same behavior. Retrospective analysis of previous projects[1481] can provide insight into common mistakes.

This section discusses the initial resource estimation process; ongoing resource estimation, performed once a project is underway, is discussed in section 5.4.4. The techniques used to produce estimate values are based on previous work that is believed to have similarities with the current project; the more well known of these techniques are discussed in section 5.3.1.

Client uncertainty about exactly what they want can have a major impact on the reliability of any estimate of the resources required. Discovering the functionality needed for acceptance is discussed in section 5.4.5.

When making an everyday purchase decision, potential customers have an expectation that the seller can, and will, provide information on cost and delivery date; an expectation of being freely given an accurate answer is not considered unreasonable. People cling to the illusion that it's reasonable to ask for accurate estimates to be made about the future (even when they have not measured the effort involved in previous projects).

Unless the client is willing to pay for a detailed analysis of the proposed project, there is no incentive for vendors to invest in a detailed analysis until a contract is signed.

While the reasons of wanting a cost estimate cannot be disputed, an analysis of the number of unknowns involved in a project, and the experience of those involved can lead to the conclusion that it is unreasonable to expect accurate resource estimates.

The largest item cost for many projects is the cost of the time of people involved. These people need to have a minimum set of required skills, and a degree of dedication; this issue is discussed in section 5.5.

Cost overruns are often blamed on poor project management,[1090] however, the estimates may be the product of rational thinking during the bidding process, e.g., a low bid swayed the choice of vendor.

People tend to be overconfident, and a cost/benefit analysis shows that within a population, individual overconfidence is an evolutionary stable cognitive bias; see section 2.8.5.

It should not be surprising that inaccurate resource estimates are endemic in many industries[1251] (and perhaps all human activities), software projects are just one instance. A study by Flyvbjerg, Holm and Buhl[611] of 258 transportation projects (worth $90 billion) found costs are underestimating in around 90% of projects, with an average cost overrun of 28% (sd 39); a study of 35 major US DOD acquisitions[215] found a 60% average growth in total costs; an analysis of 12 studies,[839] covering over 1,000 projects, found a

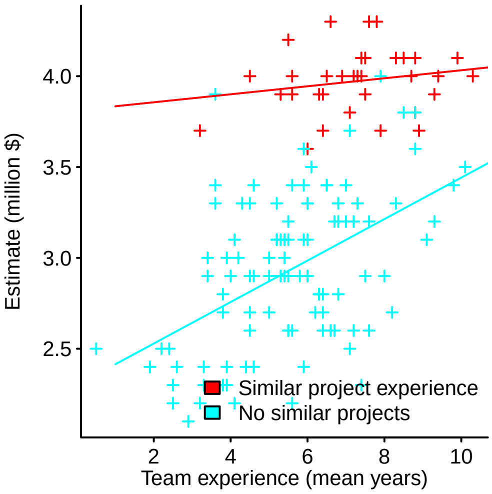


Figure 5.11: Mean number of years experience of each team against estimated project code, with fitted regression models; broken down by teams containing one or more members who have had similar project experience, or not. Data from Mcdonald.[1222] Github–Local

[iii] The desired accuracy will depend on the reason for investing in an estimate, e.g., during a feasibility study an order of magnitude value may be good enough.

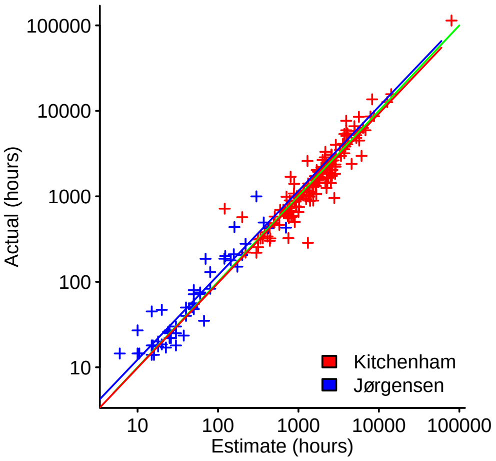


Figure 5.12: Estimated and actual project implementation effort; 49 web implementation tasks (blue), and 145 tasks performed by an outsourcing company (red). Data from Jørgensen[934] and Kitchenham et al.[1002] Github–Local

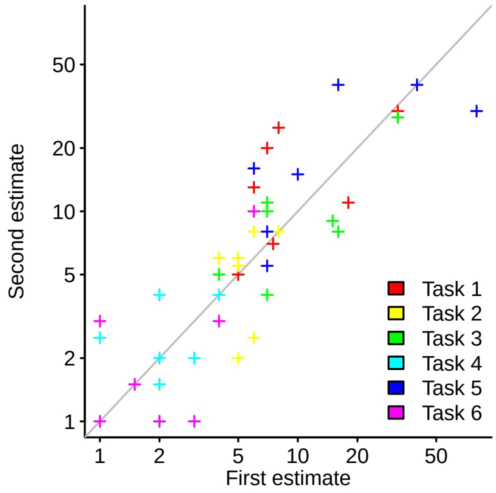


Figure 5.13: Two estimates (in work hours), made by seven subjects, for each of six tasks. Data from Grimstad et al.[734] Github–Local

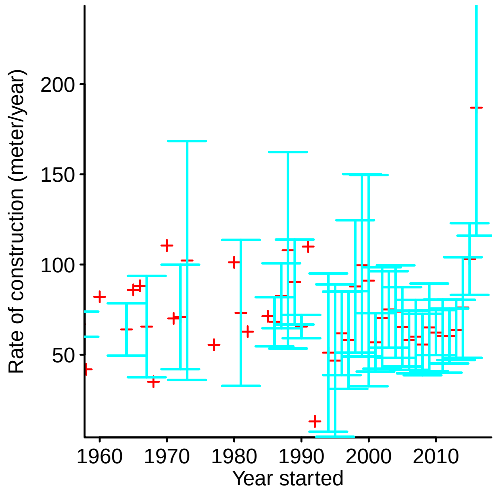


Figure 5.14: Mean rate of construction, in meters per year, of skyscrapers taller than 150 m (error bars show standard deviation). Data kindly provided by Recon.[1547] Github–Local

mean estimation error of 21%; Butts and Linton[281] give a detailed discussion of overruns on the development of over 150 NASA spacecraft.

Those working on a project may have experience from personal involvement in the implementation of other projects (organizational forgetting is discussed in section 3.4.5). The extent to which experience gained on the implementation of other projects can be used to help formulate a reliable estimate for a new project depends on shared similarities, such as: performance of the individuals involved (e.g., knowledge, skill and cognitive capacity; figure 2.36 shows that a developer can sometimes spend more time reimplementing the same specification), interactions between team members, interactions with real-world events that occur during the projects (e.g., interruptions, uncontrolled delays, client involvement).

A study by Mcdonald[1222] investigated the impact of team experience on the estimated cost of one project (which teams of professional managers had each planned for approximately 20 hours). Figure 5.11 shows the mean number of years experience of each of the 135 teams, and their estimate; teams are broken down into those having at least one member who had previous experience on a similar project, and those that did not (straight lines are fitted regression models).

Unknown and changeable factors introduce random noise into the estimation process; on average, accuracy may be good. Figure 5.12 shows regression models fitted to two estimation datasets, the green line shows where actual equals estimate. The trend for the 49 blue projects[934] is to (slightly) overestimate, while the 145 red project[1002] trend is to (slightly) underestimate.

Resource estimation is a knowledge based skill, acquired through practical experience and learning from others (expertise is discussed in section 2.5.2); an individual's attitude to risk has also been found to have an impact.[929]

A study by Grimstad and Jørgensen[734] investigated the consistency of estimates made by the same person. Seven developers were asked to estimate sixty tasks (in work hours), over a period of three months; unknown to them, everybody estimated six tasks twice. Figure 5.13 shows the first/second estimates for the same task made by the same subject; identical first/second estimates appear on the grey line, with estimates for identical tasks having the same color (the extent of color clustering shows agreement between developers).

A study by Jørgensen[935] selected six vendors, from 16 proposals received from a tender request, to each implement the same database-based system. The estimated price per work-hour ranged from \$9.1 to \$28.8 (mean \$13.85).

Skyscraper construction is like software projects, in that each is a one-off, sharing many implementation details with other skyscrapers, but also having significant unique implementation features. Figure 5.14 shows that the rate of construction of skyscrapers (in meters per year), has remaining roughly unchanged.

Some of the factors influencing the client and/or vendor resource estimation process include:

- the incentives of those making the estimate.

  When estimating in a competitive situation (e.g., bidding on a request to tender), the incentive is to bid as low as possible; once a project is underway, the client has little alternative but to pay more (project overruns receive the media attention, and so this is the more well-known case).

  When estimating is not part of a bidding process (e.g., internal projects, where those making the estimate may know the work needs to be done, and are not concerned with being undercut by competitors), one strategy is to play safe and overestimate, delivering under budget is often seen by management in a positive light,[iv]

- the cost of making the estimate.

  Is it cost effective to invest as much time estimating as it is likely to take doing the job? It depends on who is paying for the estimate, and the probability of recouping the investment in making the estimate. Unless the client is paying, time spent estimating is likely to be a small fraction of an initial crude estimate of the time needed to do the job.

[iv] The equation: $actual = estimate^{0.67}$, explains almost half the variance in the data for figure 8.17.

- estimating is a social process (see fig 2.64), people often want to get along with others, which means prior estimation information can have an impact. Client expectations can have an anchoring effect on cost estimates[945] (one study[1672] found that it was possible to reduce this effect; see Github–projects/DeBiasExptData.R).

  A study by Aranda[65] investigated the impact of anchoring on estimated task effort. Subjects (13 professionals, 10 academics) were asked to estimate how long it would take to deliver a software system specified in a document they were given. The document contained a statement from a middle manager, either expressing no opinion, or guessing that the project would take 2-months, or 20-months. Figure 5.15 shows the estimates made by subjects who saw a particular middle manager opinion (sorted to show estimate variability),
- pressure is applied to planners[1892] to ensure their analysis presents a positive case for project funding. A former president of the American Planning Association said:[610] "I believe planners and consultants in general deliberately underestimate project costs because their political bosses or clients want the projects. Sometimes, to tell the truth is to risk your job or your contracts or the next contract . . . "
- cognitive processing of numeric values can lead to different answers being given, e.g., the measurement units used (such monthly or yearly);[1843] see section 2.7.2.

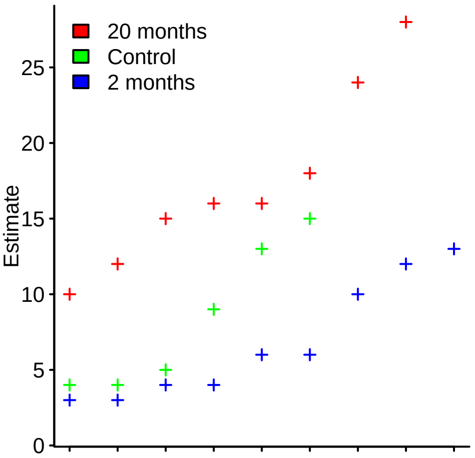


Figure 5.15: Estimate given by three groups of subjects after seeing a statement by a middle manager containing an estimate (2 months or 20 months) or no estimate (control); sorted to highlight distribution. Data from Aranda.[65] Github–Local

Projects are often divided into separate phases of development, e.g., requirements analysis, design, implementation, and testing. What is the breakdown of effort between these phases?

A study by Wang and Zhang[1904] investigated the distribution of effort, in man-hours, used during the five major phases of 2,570 projects (the types of project were: 20% new development, 68% enhancement, 7% maintenance, 5% other projects). Figure 5.16 shows a density plot of the effort invested in each phase, as a fraction of each project's total effort; the terminology in the legend follows that used by the authors (a mapping of those terms that differ from Western usage might be: Produce is implementation, Optimize is testing, and Implement is deployment). The distribution of means and medians does not vary much with project duration.

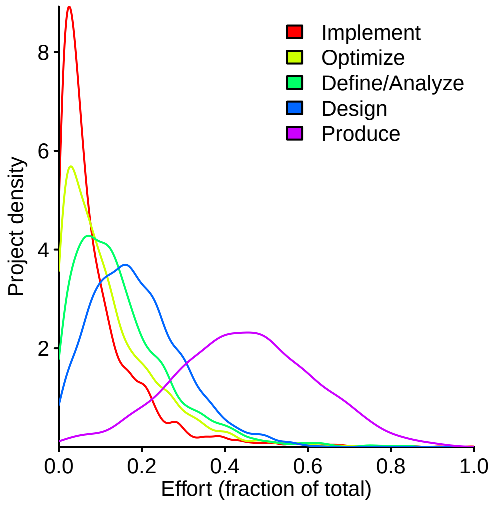


Figure 5.16: Density plot of the investment, by 2,570 projects, of a given fraction of total effort in a given project phase. Data kindly provided by Wang.[1904] Github–Local

A study by Jones and Cullum[929] investigated 8,252 agile tasks estimated and actual implementation time. Figure 5.17 shows the number of tasks having a given estimated effort and the number requiring a given actual effort. Some time estimates stand out as occurring a lot more or less frequently than nearby values, e.g., peaks at multiples of seven (there were seven hours in a work day), and very few estimates of six, eight and nine hours; the preference for certain numeric values is discussed in section 2.7.1.

In an attempt to reduce costs some companies have offshored the development of some projects, i.e., awarded the development contract to companies in other countries. A study by Šmite, Britto and van Solingen[1846] of outsourced project costs, found additional costs outside the quoted hourly rates; these were attributable to working at distance, cost of transferring the work and characteristics of the offshore site. For the projects studied, the time to likely break-even, compared to on-shore development, was thought to be several years later than planned. One study[480] attempted to calculate the offshoring costs generated by what they labeled *psychic distance* (a combination of differences including cultural, language, political, geographic, and economic development).

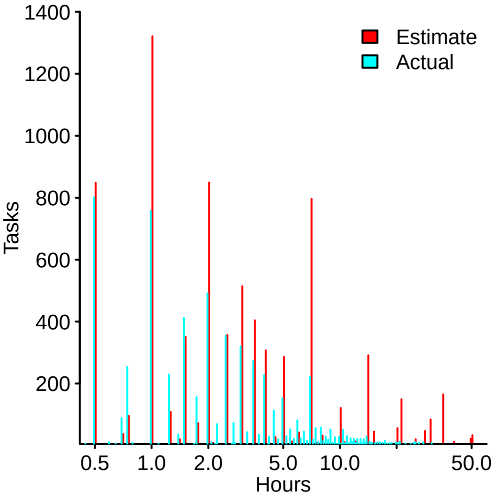


Figure 5.17: Number of tasks having a given estimate, and a given actual implementation time. Data from Jones et al.[929] Github–Local

5.3.1 Estimation models

Estimation of software development costs has proved to be a complex process, with early studies[567,568] identifying over fifty factors; a 1966 management cost estimation handbook[1344] contained a checklist of 94 questions (based on an analysis of 169 projects). Many of the existing approaches used to build estimation models were developed in the 1960s and 1970s, with refinements added over time; these approaches include: finding equations that best fit data from earlier projects, deriving and solving theoretical models, and building project development simulators.

Fitting data: Early cost estimation models fitted equations to data on past projects.[1923] The problem with fitting equations to data, is that the resulting models are only likely to perform well when estimating projects having the same characteristics as the projects from which the data was obtained. A study by Mohanty[1287] compared the estimates produced by 12 models. Figure 5.18 shows how widely the estimates varied.

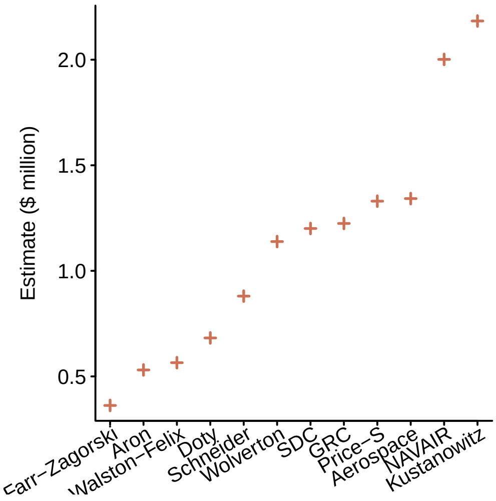


Figure 5.18: Estimated project cost from 12 estimating models. Data from Mohanty.[1287] Github–Local

A 1991 study by Ourada[1410] evaluated four effort estimation models used for military software (two COCOMO derivatives REVIC and COSTMODL, plus SASET and SEER; all fitted to data); he reached the conclusion: "I found the models to be highly inaccurate and very much dependent upon the interpretation of the input parameters." Similar conclusions were reached by Kemerer[973] who evaluated four popular cost estimation models (SLIM, COCOMO, Function Points and ESTIMACS) on data from 15 large business data processing projects; Ferens[586] compared 10 models against DOD data and came to the same conclusion, as did another study in 2003 using data from ground based systems.[771]

Current machine learning models perform estimate adjustment; they require an estimate to be given as one of the input variables, and return an adjusted value (existing public estimation data sets don't contain enough information to allow viable models to be built unless the known estimated values are used during training).

Deriving equations: In the early 1960s, Norden[1368] studied projects, which he defined as " . . . a finite sequence of purposeful, temporarily ordered activities, operating on a homogeneous set of problem elements, to meet a specified set of objectives . . . ". Completing a project involves solving a set of problems (let $W(t)$ be the proportion of problems solved at time t), and these problems are solved by the people resources ($p(t)$ encodes information on the number of people, and their skill); the rate of problems solving depends on the number of people available, and the number of problems remaining to be solved; this is modeled by the differential equation:

$$\frac{dW}{dt} = p(t)[1 - W(t)], \text{ whose solution is: } W(t) = 1 - e^{-\int^t p(\tau)d\tau}$$

If the skill of the people resource grows linearly, as the project progresses, i.e., team members learn at the rate $p(t) = at$, the work rate is:

$$\frac{dW}{dt} = ate^{-at^2/2}$$

This equation is known, from physics, as the Rayleigh curve. Putnam[1516] evangelised the use of Norden's model for large software development projects. Criticism of the Norden/Putnam model has centered around the linear growth assumption (i.e., $p(t) = at$) being unrealistic.

An analysis by Parr[1428] modeled the emergence of problems during a project as a binary tree, and assumed that enough resources are available to complete the project in the shortest possible time. Under these assumptions the derived work rate is:

$$\frac{dW}{dt} = \frac{1}{4}\text{sech}^2\frac{\alpha t + c}{2}, \text{ where sech is the hyperbolic secant: } \text{sech}(x) = \frac{2}{e^x + e^{-x}}.$$

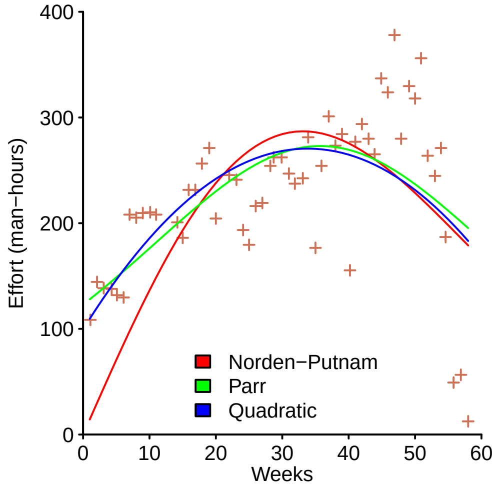


Figure 5.19: Elapsed weeks (x-axis) against effort in man-hours per week (y-axis) for a project, plus three fitted curves. Data extracted from Basili et al.[135] Github–Local

While these two equations look very different, their fitted curves are similar; one difference is that the Rayleigh curve starts at zero, while the Parr curve starts at some positive value.

A study by Basili and Beane[135] investigated the quality of fit of various models to six projects requiring around 100 man-months of effort. Figure 5.19 shows Norden-Putnam, Parr and quadratic curves fitted to effort data for project 4.

The derivation of these, or any other equation, is based on a set of assumptions, e.g., a model of problem discovery (the Norden/Putnam and Parr models assume there are no significant changes to the requirements), and the necessary manpower can be added or removed at will. The extent to which the derived equation apply to a project depends on how closely the characteristics of the project meet the assumptions used to build the model. Both the Norden-Putnam and Parr equations can be derived using hazard analysis,[1872] with the lifetime of problems to be discovered having a linear and logistic hazard rate respectively.

Simulation models: A simulation model that handles all the major processes involved in a project could be used to estimate the resources likely to be needed. There have been several attempts to build such models using Systems Dynamics.[3,269,1175] Building a simulation model requires understanding the behavior of all the important factors involved, and as the analysis in this book shows, we are a long way from having this understanding.

Subcomponent based: Breaking a problem down into its constituent parts may enable a good enough estimate to be made (based on the idea that smaller tasks are easier to understand, compared to larger, more complicated tasks), assuming there are no major interactions between components that might affect an estimate. Examples of subcomponent based estimation methods include: function-points (various function-point counting algorithms are in use), use case points and story points.

Uses case points is an estimation method that makes use of weighting factors derived from the end-user, environmental and technical complexity of a project; the appropriate weighting for over 20 factors has to be selected. A study by Ochodek, Nawrocki and Kwarciak[1382] investigated the performance of use case points for 27 projects; see Github–projects/simplifying-ucp.R.

The function point analysis methods are based on a unit of measurement, known as a *function point*, and take as input a requirements' specification or functional requirements. A formula or algorithm (they vary between the methods) is used to derive the size of each requirement in function points. The implementation cost of a function point is obtained from the analysis of previous projects, where the number of function points and costs is known.

The accuracy of function point analysis methods depends on the accuracy with which requirements are measured (in function points), and the accuracy of function point to cost mapping. Figure 11.18 suggests that the requirements measuring processing process produces consistent values.

A study by Kampstra and Verhoef[958] investigated the reliability of function point counts. Figure 5.20 shows the normalised cost for 149 projects, from one large institution, having an estimated number of function points; also see Github–projects/82507128-kitchenham and Github–projects/HuijgensPhD.R.

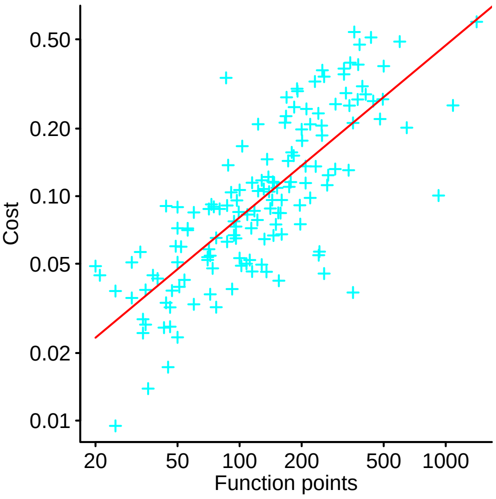


Figure 5.20: Function points and corresponding normalised costs for 149 projects from one large institution; line is a fitted regression model of the form: *Cost* ∝ *Function_Points*$^{0.75}$. Data extracted from Kampstra el al.[958] Github–Local

A study by Huijgens and van Solingen[859] investigated two projects considered to be best-in-class, out of 345 projects, from three large organizations. Figure 5.21 shows the cost per requirement, function point, and story point for these two projects over 13 releases.

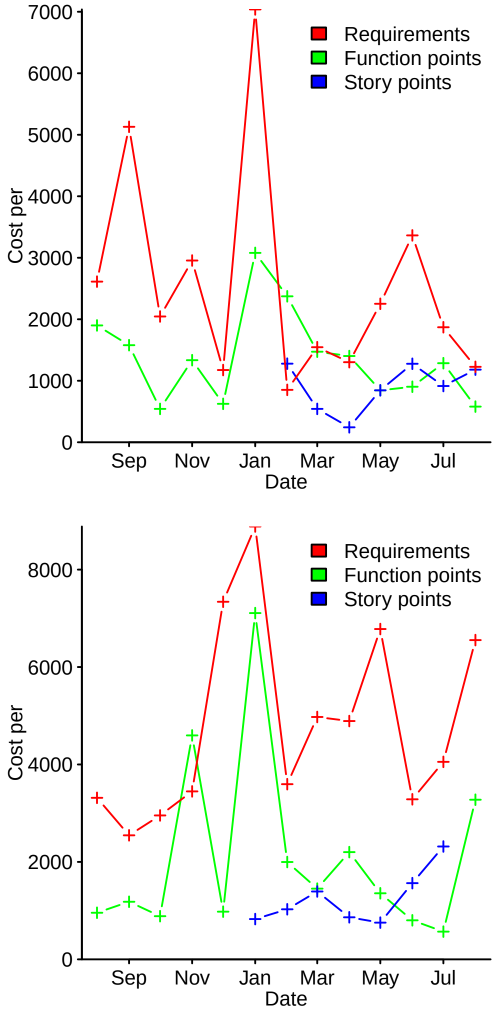


Figure 5.21: Cost per requirement, function point and story point for two projects, over 13 monthly releases. Data from Huijgens.[859] Github–Local

A study by Commeyne, Abran and Djouab[380] investigated effort estimates made using COSMIC function-points and story-points. Figure 5.22 shows the estimated number of hours needed to implement 24 story-points, against the corresponding estimated function-points.

Some estimation models include, as input, an estimate of the number of lines of code likely to be contained in the completed program (see fig 3.34). How much variation is to be expected in the lines of code contained in programs implementing the same functionality, using the same language?

Figure 5.23 shows data from seven studies, where multiple implementations of the same specification were written in the same language. The fitted regression model finds that the standard deviation is approximately one quarter of the mean. With so much variation, in LOC, between different implementations of the same specification, a wide margin of error must be assumed for any estimation technique where lines of code is a significant component of the calculation (also see fig 7.31).

Figure 9.14 shows the number of lines contained in over 6,300 C programs implementing the $3n+1$ problem. A more substantial example is provided by the five Pascal compilers targeting the same mainframe.[1676]

5.3.2 Time

When will the system be ready for use? This is the questions clients often ask immediately after the cost question (coming before the cost question can be a good sign, i.e., time is more important than cost). Many factors have been found to have a noticeable impact on the accuracy of predictions for the time needed to perform some activity.[760]

In some cases time and money are interchangeable project resources,[1110] but there may be dependencies that prevent time (or money being spent) until some item becomes available. A cost plus contract provides an incentive to spend money, with time only being a consideration if the contract specifies penalty clauses.

Figure 5.24 shows the mean and median effort spent on 2,103 projects taking a given number of elapsed months to complete; regression lines are fitted quadratic equations. Assuming 150 man-hours per month, project team size appears increase from one to perhaps six or seven, as project duration increases.

Studies[387, 943] have investigated the explanations given by estimators, for their inaccurate estimates; see Github–projects/Regression-models.R.

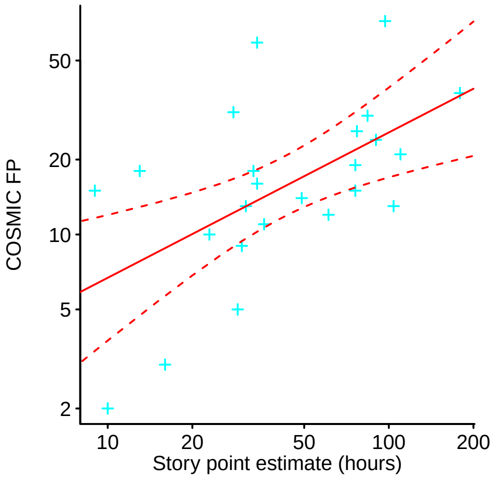


Figure 5.22: Estimated effort to implement 24 story-points and corresponding COSMIC function point; line is a fitted regression model of the form: $CosmicFP \propto storyPoint^{0.6}$, with 95% confidence intervals. Data from Commeyne et al.[380] Github–Local

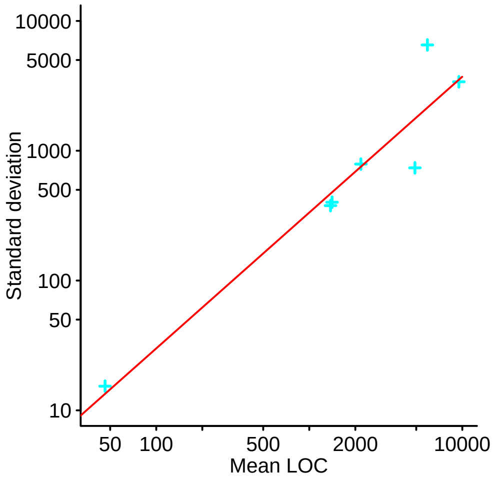


Figure 5.23: Mean LOC against standard deviation of LOC, for multiple implementations of seven distinct problems; line is a fitted regression model of the form: $Standard\_deviation \propto SLOC$. Data from: Anda et al,[52] Jørgensen,[935] Lauterbach,[1085] McAllister et al,[1212] Selby et al,[1649] Shimasaki et al,[1676] van der Meulen.[1857] Github–Local

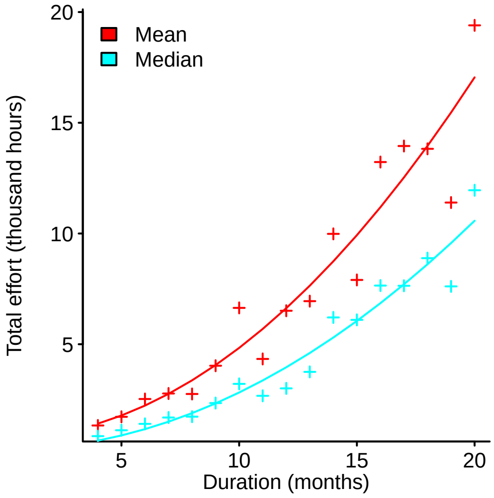


Figure 5.24: Mean and median effort (thousand hours) for projects having a given elapsed time; both lines are a fitted regression model of the form: $Effort \propto Duration^2$. Data from Wang et al.[1904] Github–Local

5.3.3 Size

Program size is an important consideration when computer memory capacity is measured in kilobytes. During the 1960s, mainframe computer memory was often measured in kilobytes, as was minicomputer memory during the 1970s, and microcomputer memory during the 1980s. Today, low cost embedded controllers might contain a kilobyte of memory, or less, e.g., electronic control units (ECU) used in car engines contain a few kilobytes of flash memory.

Pricing computers based on their memory capacity is a common sales technique. Figure 5.25 shows IBM's profit margin on sales of all System 360s sold in 1966 (average monthly rental cost, for 1967, in parentheses). Low-end systems, with minimal memory, were sold below cost to attract customers (and make it difficult for competitors to gain a foothold in the market[472]).

The practices adopted to handle these memory size constraints continue to echo in software engineering folklore.

A study by Lind and Heldal[1139] investigated the possibility of using COSMIC function points to estimate the compiled size of software components used in ECUs. Function points were counted for existing software components in the ECUs used by Saab and General Motors. Figure 5.26 shows COSMIC function points against compiled size of components from four ECU modules; lines are fitted regression models for each module. While a consistent mapping exists for components within each module, the function point counting technique used did not capture enough information to produce consistent results across modules.

A study by Prechelt[1503] investigated a competition in which nine teams, each consisting of three professional developers, were given 30 hours to implement 132 functional and 19 non-functional requirements, for a web-based system; three teams used Java, three used Perl, and three used PHP (two teams used Zend, and every other team chose a different framework).

Figure 5.27 shows how the number of lines of manually written code, and the number of requirements implemented, varied between teams; many more significant differences between team implementations are discussed in the report.[1503]

5.4 Paths to delivery

There are many ways of implementing a software system, and those involved have to find one that can be implemented using the available resources.

The path to delivery may include one or more pivots, where a significant change occurs.[116] Pivots can vary from fundamental changes (e.g., involving the target market, or the client's requirements), to technical issues over which database to use.

Creating a usable software system involves discovering a huge number of interconnected details;[1309] these details are discovered during: requirements gathering to find out what the client wants (and ongoing client usability issues during subsequent phases[1187]), formulating a design[416] for a system having the desired behavior, implementing the details and their interconnected relationships in code, and testing the emergent behavior to ensure an acceptable level of confidence in its behavior (testing is covered in chapter 6).

Managing the implementation of a software system is an exercise in juggling the available resources, managing the risks (e.g., understanding what is most likely to go wrong,[99] and having some idea about what to do about it), along with keeping the client convinced[v] that a working system will be delivered within budget and schedule; these are all standard project management issues.[1008] Managements interest in technical issues focuses on the amount of resource they are likely to consume, and the developer skill-sets needed to implement them.

Figure 5.28 shows an implementation schedule for one of the projects studied by Ge and Xu.[655] This plot gives the impression of a project under control, in practice schedules evolve, sometimes beyond all recognition. As work progresses, discovered information can result in schedule changes, e.g., slippage on another project can prevent scheduled

[v] Successful politicians focus on being elected, and then staying in office:[445] successful managers focus on getting projects, and then keeping the client on-board; unconvinced clients cancel projects, or look for others to take it over.

staff being available, and a requirement change creates a new dependency that prevents some subcomponents being implemented concurrently;

Large projects sometimes involve multiple companies, and the initial division of work between them may evolve over time.

A study by Yu[1975] investigated the development of three multi-million pound projects. The chronology of events documented provides some insight into how responsibility for the implementation of functionality, and associated costs, can shift between project contractors as new issues are uncovered, and contracted budgets are spent. Figure 5.29 shows the changes in contractors' project cost estimates, for their own work, over the duration of the project.

Vendors want to keep their clients happy, but not if it means loosing lots of money. A good client relationship manager knows when to tell the client that extra funding is needed to support the requested change, and when to accept it without charging (any connection with implementation effort is only guaranteed to occur for change requests having significant cost impact).

Daily client contact has been found to have an impact on final project effort; see fig 11.44. The study did not collect data on the extent to which the planned project goals were achieved, and it is possible that daily contact enabled the contractor to convince the client to be willing to accept a deliverable that did not support all the functionality originally agreed.

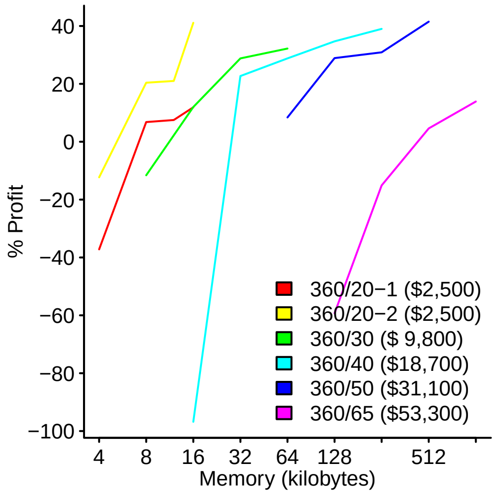


Figure 5.25: IBM's profit margin on all System 360s sold in 1966, by system memory capacity in kilobytes; monthly rental cost during 1967 in parentheses. Data from DeLamarter.[472] Github–Local

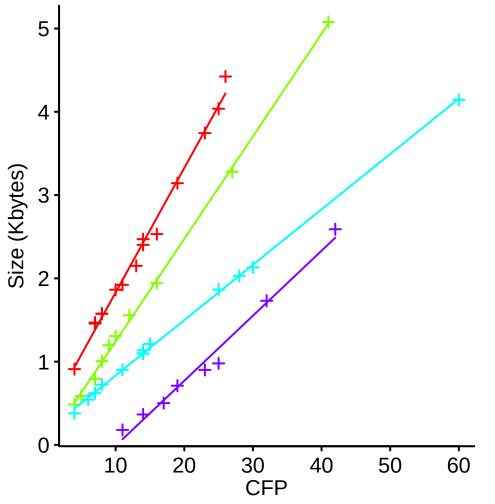


Figure 5.26: COSMIC function-points and compiled size (in kilobytes) of components in four different ECU modules; lines show fitted regression model. Data from Lind et al.[1139] Github–Local

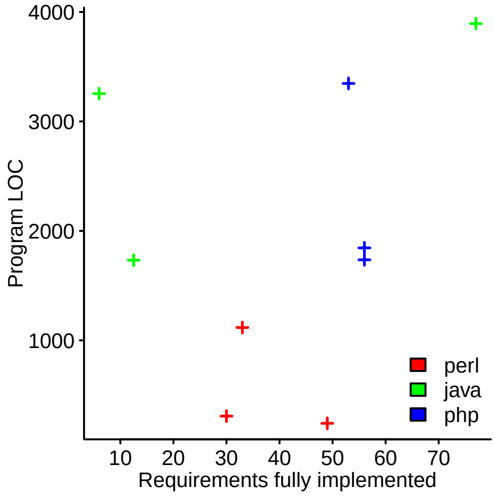


Figure 5.27: Number of requirements and corresponding lines of manually created source code, for each team (colors denote language used). Data from Prechelt[1503] Github–Local

5.4.1 Development methodologies

Software development is a punctuated information arrival process. [vi] It took several decades before a development methodology designed to handle this kind of process started to be widely used.

The first development methodology,[691] written in 1947, specified a six-step development process, starting with the people who performed the high-level conceptual activities (i.e., the scientists and engineers), and ending with the people who performed what was considered to be the straightforward coding activity (performed by the coders). Over time, the complexity of development has resulted in a significant shift of implementation authority, and power, to those connected with writing the code.[540]

A study by Rico,[1560] going back over 50-years from 2004, identified 32 major techniques (broken down by hardware era and market conditions) that it was claimed would produce savings in development or maintenance. There have been a few field studies of the methodologies actually used by developers (rather than what managers claim to be using); system development methodologies have been found to provide management support for necessary fictions,[1333] e.g., a means to creating an image of control to the client, or others outside the development group.

The *Waterfall model* continues to haunt software project management, despite repeated exorcisms over several decades.[1075] The paper[1592] that gave birth to this demon meme contained a diagram, with accompanying text claiming this approach was risky and invited failure, with diagrams and text on subsequent pages showing the recommended approach to producing the desired outcome. The how not to do it approach, illustrated in the first diagram, was taken up, becoming known as the waterfall model, and included in the first version of an influential standard, DoD-Std-2167,[475] as the recommended project management technique.[vii] Projects have been implemented using a Waterfall approach.[1955]

Iterative development has been independently discovered many times.[1075]

The U.S. National Defense Authorization Act, for fiscal year 2010, specified that an iterative acquisition process be used for information technology systems; Section 804:[1164] contains the headline requirements "(B) multiple, rapidly executed increments or releases of capability; (C) early, successive prototyping to support an evolutionary approach; . . . ". However, the wording used for the implementation of the new process specifies: " . . . well-scoped and well-defined requirements.", i.e., bureaucratic inertia holds sway.

[vi] Particular development activities may have dominant modes of information discovery.

[vii] Subsequent versions removed this recommendation, and more recent versions recommend incremental and iterative development. The primary author of DoD-Std-2167 expressed regret for creating the strict waterfall-based standard, that others advised him it was an excellent model; in hindsight, had he known about incremental and iterative development, this would have been strongly recommended, rather than the waterfall model.[1075]

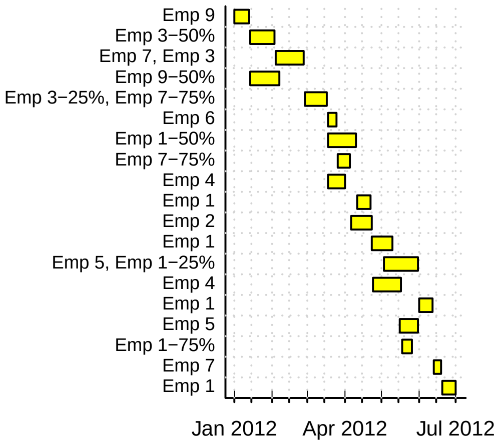


Figure 5.28: Initial implementation schedule, with employee number(s) given for each task (percentage given when not 100%) for a project. Data from Ge et al.[655] Github–Local

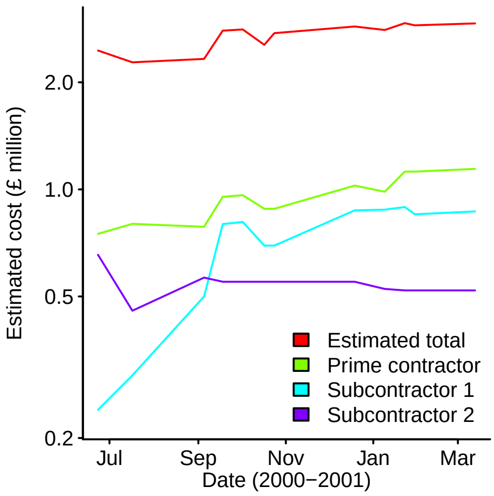


Figure 5.29: Evolution of the estimated cost of developing a bespoke software system, as implementation progressed; over time the estimated costs shift between the Prime contractor and its two subcontractors. Data from Yu.[1975] Github–Local

The battle fought over the communication protocols used to implement the Internet is perhaps the largest example of a waterfall (the OSI seven-layer model, ISO 7498 or ITU-T X.200, documented the requirements first, which vendors could then implement, but few ever did) vs. an iterative approach (the IETF process was/is based on rough consensus and running code,[1599] and won the battle).

The economic incentives and organizational structure in which a project is developed can be a decisive factor in the choice of development methodology. For instance, the documentation and cost/schedule oversight involved in large government contracts requires a methodology capable of generating the necessary cost and schedule documents; in winner take-all markets, there is a strong incentive to be actively engaging with customers as soon as possible, and a methodology capable of regularly releasing updated systems provides a means of rapidly responding to customer demand, as well as reducing the delay between writing code and generating revenue from it.

The choice of project implementation strategy is strongly influenced by the risk profile of changes to requirements and company cash flow.

The benefits of using iterative development include: not requiring a large upfront investment in requirements gathering, cost reduction from not implementing unwanted requirements (based on feedback from users of the current system), working software makes it possible to start building a customer base (a crucial requirement in winner take-all markets), and reducing the lag between writing code and earning income from it.

The additional cost incurred from using iterative development, compared to an upfront design approach, come from having to continually rearchitect programs to enable them to support functionality that they were not originally designed to support.

If the cost of modifying the design is very high, it can be cost effective to do most of the design before implementing it. The cost of design changes may be high when: hardware that cannot be changed is involved, or when many organizations are involved and all need to work to the same design.

When customer requirements are rapidly changing, or contain many unknowns, the proposed design may quickly become out-of-date, and it may be more cost effective to actively drive the design by shipping something working (e.g., evolving a series of languages, and their corresponding compilers;[138] see Github–projects/J04.R). User feedback is the input to each iteration, and without appropriate feedback iterative projects can fail.[715]

The growth of the Internet provided an ideal ecosystem for the use of iterative techniques. With everything being new, and nobody knowing what was going to happen next: requirements were uncertain and subject to rapid change, many market niches had winner-take-all characteristics. Also, the Internet provided a distribution channel for frequent software updates.

Project lifespan by be sufficiently uncertain that it dictates the approach to system development. For instance, building the minimum viable product, and making it available to potential customers to find out if enough of them are willing to pay enough for it to be worthwhile investing in further development.

In an environment where projects are regularly cancelled before delivery, or delivered systems have a short lifespan, it may be a cost effective to make short term cost savings that have a larger long term cost.

A study by Özbek[1413] investigated attempts to introduce software engineering innovations into 13 open source projects within one-year. Of the 83 innovations identified in the respective project email discussions 30 (36.1%) were successful, 37 (44.6%) failed, and 16 (19.3%) classified as unknown.

5.4.2 The Waterfall/iterative approach

The major activities involved in the waterfall/iterative approach to building a software system include: requirements, design, implementation (or coding), testing and deployment. These activities have a natural ordering in the sense that it is unwise to deploy without some degree of testing, which requires code to test, which ideally has been designed and implements a known requirement. The extent to which documentation is a major activity, or an after-thought, depends on the client.

The term _phase</phase> is sometimes used to denote project activities, implying both an ordering, and a distinct period during which one activity is performed.

A study by Zelkowitz[1983] investigated when work from a particular activity was performed, relative to other activities (for thirteen projects, average total effort 13,522 hours). Figure 5.30 shows considerable overlap between all activity, e.g., 31.3% of the time spent on design occurred during the coding and testing activity; also see Github–projects/zelkowitz-effect.R.

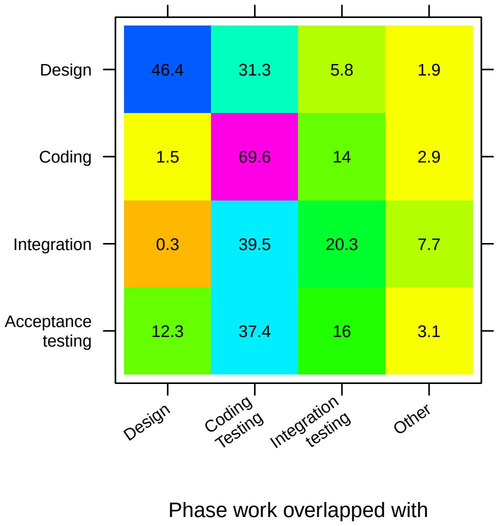
How are the available resources distributed across the major project activities?

A study by Condon, Regardie, Stark and Waligora[385] investigated 39 applications developed by the NASA Flight Dynamics Division between 1976 and 1992. Figure 5.31 shows ternary plots of the percentage effort, in time (red), and percentage schedule, in elapsed time (blue), for design, coding and testing (mean percentages for the three activities were: effort time: 24%, 43 and 33 respectively and schedule elapsed time: 33%, 34 and 33).

To what extent does resources distribution across major project activities vary between organizations?

Figure 5.32 shows a ternary plot for the design/coding/test effort for projects developed by various organizations: a study by Kitchenham and Taylor[1003] investigated a computer manufacturer (red) and a telecoms company (green), a study by Graver, Carriere, Balkovich and Thibodeau[721] investigated space projects (blue) and major defense systems (purple).

There is a clustering of effort breakdowns for different application areas; the mean percentage design, coding and testing effort were: computer/telecoms (17%, 57, 26) and Space/Defence (36%, 20, 43). There is less scope to recover from the failures of software systems operating in some space/defense environments; this operational reality makes it worthwhile investing more in design and testing.



	Design	Coding Testing	Integration testing	Other
Design	46.4	31.3	5.8	1.9
Coding	1.5	69.6	14	2.9
Integration	0.3	39.5	20.3	7.7
Acceptance testing	12.3	37.4	16	3.1

Figure 5.30: Phase during which work on a given activity of development was actually performed, average percentages over 13 projects. Data from Zelkowitz.[1983] Github–Local

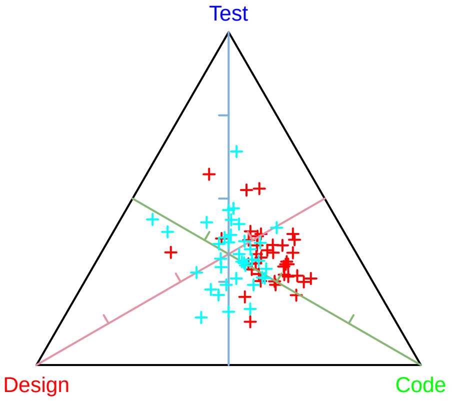


Figure 5.31: Percentage distribution of effort time (red) and schedule time (blue) across design/coding/testing for 38 NASA projects. Data from Condon et al.[385] Github–Local

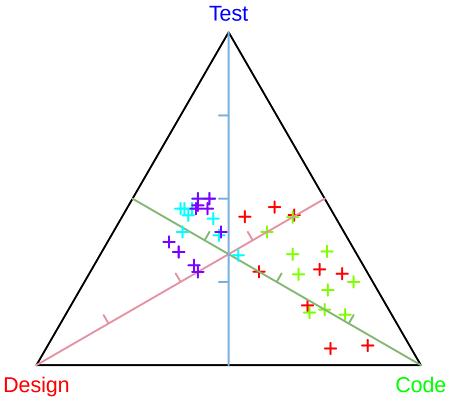


Figure 5.32: Percentage distribution of effort across design/coding/testing for 10 ICL projects (red), 11 BT projects (green), 11 space projects (blue) and 12 defense projects (purple). Data from Kitchenham et al[1003] and Graver et al.[721] Github–Local

5.4.3 The Agile approach

The Agile manifesto specified "Individuals and interactions over processes and tools", but this has not stopped the creation and marketing of a wide variety of processes claiming to be the agile way of doing things. The rarity of measurement data for any of the agile processes means this evidence-based book has little to say about them.

5.4.4 Managing progress

How is the project progressing, in implemented functionality, cost and elapsed time, towards the target of a project deliverable that is accepted by the client?

A project involves work being done and people doing it. Work and staff have to be meshed together within a schedule; project managers will have a view on what has to be optimized. A company employing a relatively fixed number of developers may seek to schedule work so that employees always have something productive to do, while a company with a contract to deliver a system by a given date may seek to schedule staff to optimise for delivery dates (subject to the constraints of any other projects).

Scheduling a software project involves solving many of the kinds of problems encountered in non-software projects, e.g., staff availability (in time and skills), dependencies between work items[233] and other projects, and lead time on the client making requirements' decisions.[1953] Common to engineering projects in general,[206] issues include the difficulty of finding people with the necessary skills, and high turnover rate of current staff; staffing is discussed in section 5.5.

Analyzing project progress data in isolation can be misleading. A study by Curtis, Sheppard and Kruesi[417] investigated the characteristics (e.g., effort and mistakes found) of the implementation of five relatively independent subcomponents of one system. Figure 5.33 shows how effort, in person hours, changes as the subcomponent projects progressed. While the five subcomponents were implemented by different teams, it is not known whether teams members worked on other projects within the company. Across multiple ongoing projects, total available effort may not change significantly, but large changes can occur on single projects (because senior management are optimizing staffing across all projects; see fig 5.4).

While it may be obvious to those working on a project that the schedule will not be met, nobody wants to be the bearer of bad news, and management rarely have anything to gain

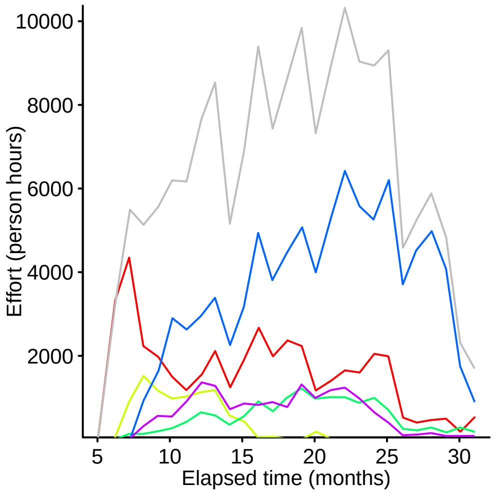


Figure 5.33: Effort, in person hours per month, used in the implementation of the five components making up the PAVE PAWS project (grey line shows total effort). Data extracted from Curtis et al.[417] Github–Local

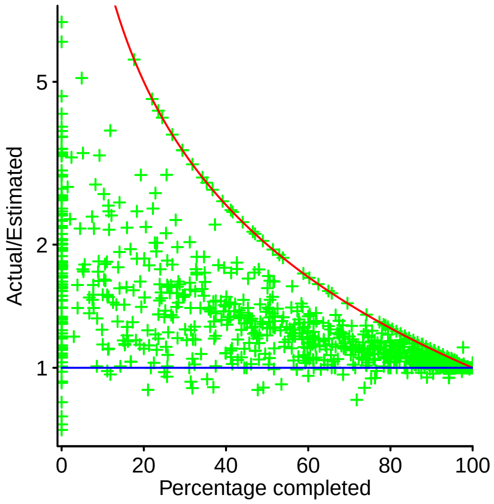


Figure 5.34: Percentage of actual project duration elapsed at the time 882 schedule estimates were made, during 121 projects, against estimated/actual time ratio (y-axis has a log scale; boundary maximum in red). Data kindly provided by Little.[1141] Github–Local

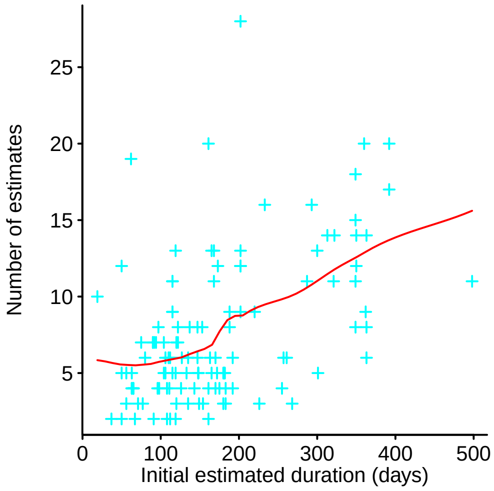


Figure 5.35: Initial estimated project duration against number of schedule estimates made before completion, for 121 projects; line is a loess fit. Data kindly provided by Little.[1141] Github–Local

by reporting bad news earlier than they have to. Progress reports detailing poor progress, and increased costs, may be ignored,[971] or trigger an escalation of commitment.[202]

Once the client believes the estimated completion date and/or cost is not going to be met, either: the project objectives are scaled back, the project cancelled, or a new completion estimate is accepted. Scheduling when to tell clients about project delivery slippage, and/or a potential cost overrun, is an essential project management skill.

The scheduling uncertainty around a successful project is likely to decrease as it progresses towards completion. The metaphor of a *cone of uncertainty* is sometimes used when discussing project uncertainty; this metaphor is at best useless. The cone shaped curve(s) that are sometimes visible in plots where the y-axis is the ratio of actual and estimated, and the x-axis is percentage completed (a quantity that is unknown until project completion, i.e., the duration of the project), are a mathematical artefact. Figure 5.34 shows a plot of percentage actually completed against the ratio $\frac{Actual}{Estimated}$, for each corresponding project (4.6% of estimate completion dates are less than the actual date), for 882 estimates made during the implementation of 121 projects; the curved boundary is a mathematical artefact created by the choice of axis, i.e., the following holds: $\frac{Actual}{Estimated} \le x_{percentage}Actual$, cancelling *Actual* gives: $\frac{1}{Estimated} \le x_{percentage}$, i.e., whatever the value of *Estimated*, the plotted point always appears below, or on, the $\frac{1}{x}$ curve.

A study by Little[1141] investigated schedule estimation for 121 projects at Landmark Graphics between 1999 and 2002. An estimated release date was made at the start, and whenever the core team reached consensus on a new date (averaging 7.2 estimates per project; Figure 5.35 shows the number of release date estimates made for 121 projects, for a corresponding initial estimated project duration, on the x-axis). The extent to which the schedule estimation characteristics found in Landmark projects are applicable to other companies will depend on issues such as corporate culture and requirements volatility. The Landmark Graphics corporate culture viewed estimates as targets, i.e., “what's the earliest date by which you can't prove you won't be finished?”;[1141] different schedule characteristics are likely to be found in a corporate culture that punishes the bearer of bad news.

Reasons for failing to meet a project deadline include (starting points for lawyers looking for strategies to use, to argue their client's case after a project fails[1576]): an unrealistic schedule, a failure of project management, changes in the requirements, encountering unexpected problems, staffing problems (i.e., recruiting or retaining people with the required skills), and being blocked because a dependency is not available.[233] Missed deadlines are common, and a common response is to produce an updated schedule.

If a project is unlikely to meet its scheduled release date, those paying for the work have to be given sufficient notice about the expected need for more resources, so the resources can be made available (or not).

Figure 5.36 shows 882 changed delivery date announcements, with percentage elapsed time when the estimate was announced along the x-axis (based on the current estimated duration of the project), and percentage change in project duration (for the corresponding project) along the y-axis; red line is a loess fit. On average larger changes in duration occur near the start of projects, with smaller changes made towards the estimated end date. The blue line is a density plot of the percentage elapsed time of when schedule change announcements are made (density values not shown). There is a flurry of new estimates after a project starts, but over 50% of all estimates are made in the last 71% of estimated remaining time, and 25% in the last 6.4% of remaining time.

Parkinson's law says that work expands to fill the time available. When management announces an estimated duration for a project, it is possible that those involved choose to work in a way that meets the estimate (assuming it would have been possible to complete the project in less time).

A study by van Oorschot, Bertrand and Rutte[1860] investigated the completion time of 424 work packages involving advanced micro-lithography systems, each having an estimated lead time. Figure 5.37 shows the percentage of work packages having a given management estimated lead time that are actually completed within a given amount of time, with colored lines showing stratification by management estimate.

People sometimes strive to meet a prominent deadline.

A study by Allen, Dechow, Pope and Wu[32] investigated Marathon finishing times for nine million competitors. Figure 5.38 shows the number of runners completing a Marathon in a given number of minutes, for a sample of 250,000 competitors. Step changes in completion times are visible at 3, 3.5, 4, 4.5, and 5 hour finish times.

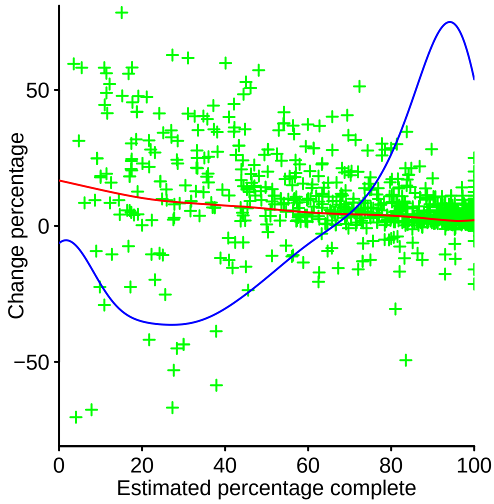


Figure 5.36: Percentage change in 882 estimated delivery dates, announced at a given percentage of the estimated elapsed time of the corresponding project, for 121 projects (red is a loess fit); blue line is a density plot of percentage estimated duration when the estimate was made. Data kindly provided by Little.[1141] Github–Local

5.4.5 Discovering functionality needed for acceptance

What functionality does a software system need to support for a project delivery to be accepted[viii] by the client?

Requirements gathering is the starting point of the supply chain of developing bespoke software, and is an iterative process.[332]

Bespoke software development is not a service that many clients regularly fund, and they are likely to have an expectation of agreeing costs and delivery dates for agreed functionality, before signing a contract.

The higher the cost of failing to obtain good enough information on the important requirements, the greater the benefit from investing to obtain more confidence that all the important requirements are known in good enough detail. Building a prototype[1709] can be a cost-effective approach to helping decide and refine requirements, as well as evaluating technologies. Another approach to handling requirement uncertainty is to build a minimum viable prototype, and then add features in response to feedback from the customer, e.g., using an agile process.

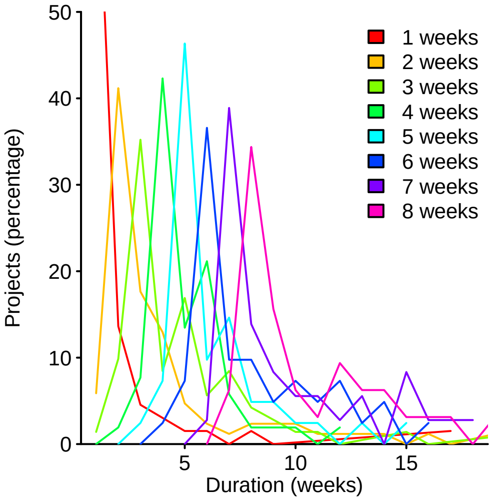


Figure 5.37: Percentage of work packages having a given lead time that are completed within a given duration; colored lines are work packages having the same estimated lead time. Data extracted from van Oorschot et al.[1860] Github–Local

The *requirements gathering* process (other terms used include: *requirements elicitation*, *requirements capture* and *systems analysis*) is influenced by the environment in which the project is being developed:

- when an existing manual way of working, or computer system, is being replaced (over half the projects in one recent survey[936]), the stakeholders of the existing system are an important source of requirements information,
- when the software is to be sold to multiple customers, as a product, those in charge of the project select the requirements. In this entrepreneurial role, the trade-off involves minimising investment in implementing functionality against maximising expected sales income,
- when starting an open source project the clients are those willing to do the work, or contribute funding.

The client who signs the cheques is the final authority on which requirements have to be implemented, and their relative priority. Clients who fail to manage the requirements process end up spending their money on a bespoke system meeting somebody else's needs.[1498]

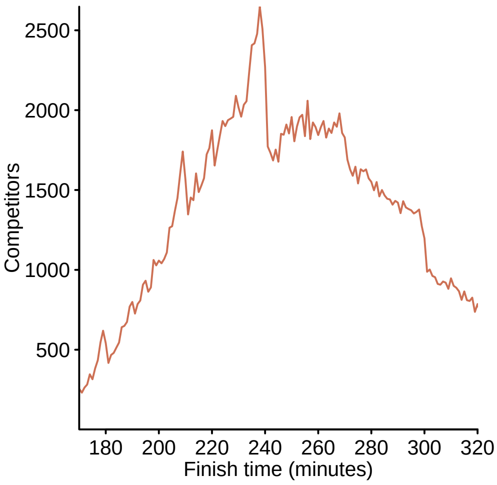


Figure 5.38: Number of Marathon competitors finishing in a given number of minutes (250,000 runner sample size). Data from Allen et al.[32] Github–Local

It is not always obvious whether a requirement has been met;[1503] ambiguity in requirement specifications is discussed in chapter 6. Sometimes existing requirements are modified on the basis of what the code does, rather than what the specification said it should do.[269] Gause and Weinberg[651] provide a readable tour through requirements handling in industry.

What is a cost effective way of discovering requirements, and their relative priority?

A *stakeholder* is someone who gains or loses something (e.g., status, money, change of job description) as a result of a project going live. Stakeholders are a source of political support, resistance to change,[1891] and active subversion[1586] (i.e., doing what they can to obstruct progress on the project), they may provide essential requirements' information, or may be an explicit target for exclusion (e.g., criminals with an interest in security card access systems).

Failure to identify the important stakeholders can result in missing or poorly prioritized requirements, which can have a significant impact on project success. What impact might different stakeholder selection strategies have?

A study by Lim[1131] investigated the University College London project to combine different access control mechanisms into one, e.g., access to the library and fitness centre. The Replacement Access, Library and ID Card (RALIC) project had been deployed two years before the study started, and had more than 60 stakeholder groups. The project documentation, along with interviews of those involved (gathering data after project completion

[viii] Acceptance here means paying all the money specified in the contract, plus any other subsequently agreed payments.

means some degree of hindsight bias will be present), was used to create the *Ground truth* of project stakeholder identification (85 people), their rank within a role, requirements and relative priorities.

The following two algorithms were used to create a final list of stakeholders:

- starting from an initial list of 22 names and 28 stakeholder roles, four iterations of Snowball sampling resulted in a total of 61 responses containing 127 stakeholder names and priorities, and 70 stakeholder roles (known as the *Open list*),
- a list of 50 possible stakeholders was created from the project documentation. The people on this list were asked to indicate which of those names on the list they considered to be stakeholders, and to assign them a salience[ix] between 1 and 10, they were also given the option to suggest other names as possible stakeholders. This process generated a list containing 76 stakeholders names and priorities, and 39 stakeholder roles (known as the *Closed list*).

How might a list of people, and the salience each of them assigns to others, be combined to give a single salience value for each person?

Existing stakeholders are in a relationship network. Lim assumed that the rank of stakeholder roles, calculated using social network metrics, would be strongly correlated with the rank ordering of stakeholder roles in the Grounded truth list. Figure 5.39 shows the Open and Closed stakeholder aggregated salience values, calculated using Pagerank (treating each stakeholder as a node in the network created using the respective lists; Pagerank was chosen for this example because it had one of the strongest correlations with the Ground truth ranking). Also, see Github–projects/requirements/stake-ground-cor.R.

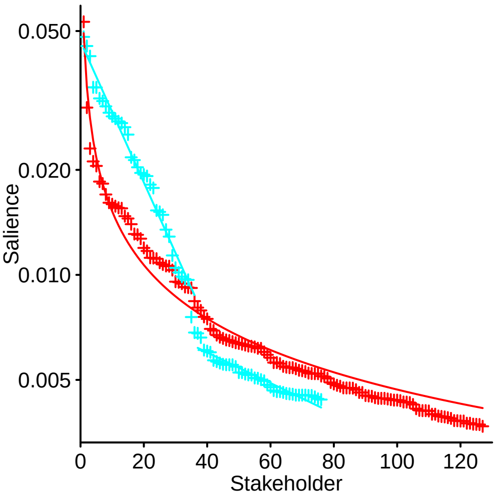


Figure 5.39: Aggregated salience of each stakeholder, calculated using the pagerank of the stakeholders in the network created from the Open (red) and Closed (blue) stakeholder responses (values for each have been sorted). Data from Lim.[1131] Github–Local

Identifying stakeholders and future users is just the beginning. Once found, they have to be convinced to commit some of their time to a project they may have no immediate interest in; stakeholders will have their own motivations for specifying requirements. When the desired information is contained in the heads of a few key players, these need to be kept interested, and involved throughout the project. Few people are practiced in requirements' specification, and obtaining the desired information is likely to be an iterative process, e.g., they describe solutions rather than requirements.

Prioritization: clients will consider some requirement to be more important than others; concentrating resources on the high priority requirements is a cost-effective way of keeping the client happy, and potentially creating a more effective system (for the resources invested). Techniques for prioritising requirements include:

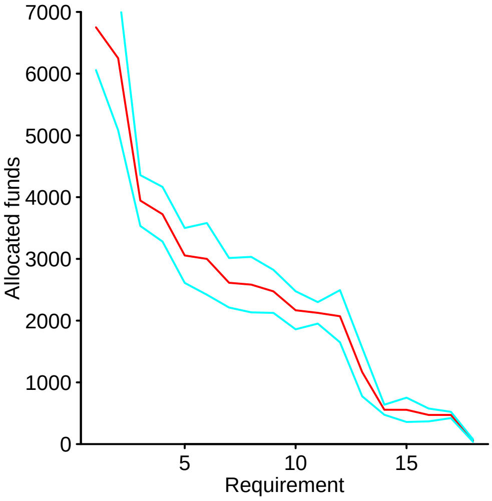


Figure 5.40: Average value assigned to requirements (red) and one standard deviation bounds (blue) based on omitting one stakeholder's priority value list. Data from Regnell et al.[1548] Github–Local

- aggregating the priority list: this involves averaging stakeholders' list of priority values, possibly weighting by stakeholder.

  To what extent are the final requirements' priorities dependent on one stakeholder? Calculating an average, with each stakeholder excluded in turn, is one method of estimating the variance of priority assignments.

  A study by Regnell, Höst, Dag, Beremark and Hjel[1548] asked each stakeholder/subject to assign a value to every item on a list of requirements, without exceeding a specified maximum amount (i.e., to act as-if they had a fixed amount of money to spend on the listed requirements). Figure 5.40 shows the average value assigned to each requirement, and the standard deviation in this value when stakeholders were excluded, one at a time.

- performing a cost/benefit analysis on each requirement, and prioritizing based on the benefits provided for a given cost;[965] see Github–projects/requirements/cost-value.R.

How much effort might need to be invested to produce a detailed requirements' specification? One effort estimate[919] for the writing of the 1990 edition of the C Standard is 50 person years, a 219-page A4 document; the effort for the next version of the standard, C99 (a 538-page document), was estimated to be 12 person years.

The development of a new product will involve the writing of a User manual. There are benefits to writing this manual first, and treating it as a requirements' specification.[185]

When a project makes use of a lot of existing source code, it may be necessary to retrofit requirements, i.e., establish traceability between existing code that is believed to implement another set of requirements. It is sometimes possible to reverse engineer links from existing code to a list of requirements.[1013]

[ix] *Salience* is defined as the degree to which managers give priority to competing stakeholder claims.[1277]

5.4.6 Implementation

The traditional measures of implementation activity are staffing (see section 5.5), and lines of code produced; management activities don't seem to have attracted any evidence-based research. Implementation activities include:[1254, 1503, 1778, 1790] meetings, design, coding, waiting for other work to be ready to use, and administration related activities; see Github–projects/E100.D_2016-TaskLog.R.

Figure 5.10 shows that as the size of a project increases, the percentage of project effort consumed by management time rapidly increases to a plateau, with fixed-price contracts involving a greater percentage of management time.

Issues around the source code created during implementation are discussed in chapter 7, and issues involving the reliability of the implementation are discussed in chapter 6.

The assorted agile methodologies include various implementation activities that can be measured, e.g., number and duration of sprints, user stories and features.

How might patterns in agile implementation activities be used to gain some understanding of the behavior of processes that are active during implementation?

7digital[1] is a digital media delivery company using an agile process to develop an open API platform providing business to business digital media services; between April 2009 and July 2012, 3,238 features were implemented by the development team (this data was kindly made available).

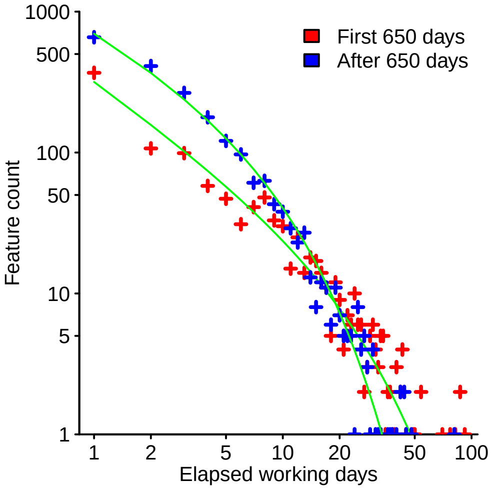


Figure 5.41: Number of features whose implementation took a given number of elapsed workdays; red first 650-days, blue post 650-days, green lines are fitted zero-truncated negative binomial distributions. Data kindly supplied by 7Digital.[1] Github–Local

Figure 5.41 shows the number of features requiring a given number of elapsed working days for their implementation (red first 650-days, blue post 650-days); a zero-truncated negative binomial distribution is a good fit to both sets of data (green lines). One interpretation for the fitted probability distribution is that there are many distinct processes involved in the implementation of a feature, with the duration of each process being a distinct Poisson process; a sum of independent Poisson processes can produce a Negative Binomial distribution. In other words, there is no single process dominating implementation time; improving feature delivery time requires improving many different processes (the average elapsed duration to implement a feature has decreased over time).

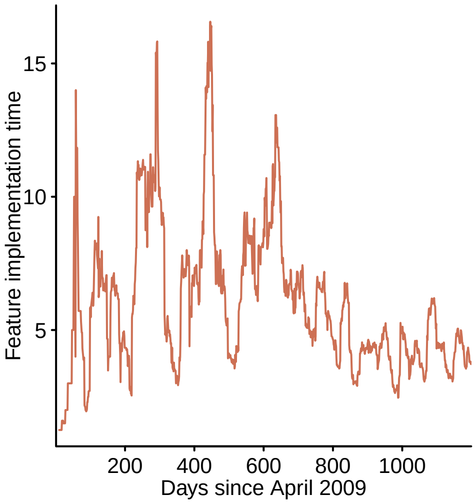


Figure 5.42: Average number of days taken to implement a feature, over time; smoothed using a 25-day rolling mean. Data kindly supplied by 7Digital.[1] Github–Local

The same distribution being a good fit for both the pre and post 650-day implementation time suggests that changes in behavior were not a fundamental, but akin to turning a dial on the distribution parameters, one-way or the other (the first 650-days has a greater mean and variance than post 650-days). If the two sets of data were better fitted by different distributions, the processes generating the two patterns of behavior are likely to have been different.

Why was a 650-days boundary chosen? Figure 5.42 shows a time series of the feature implementation time (smoothed using a 25-day rolling average). The variance in average implementation time has a change-point around 650 days (a change-point in the mean occurs around 780 days).

Figure 5.43 shows the number of new features and number of bug fixes started per day (smoothed using a 25-day rolling mean).

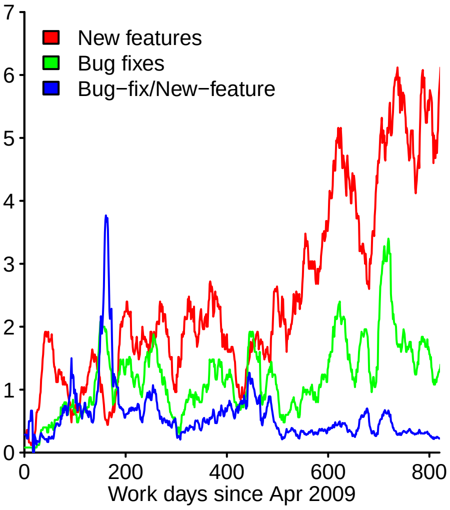


Figure 5.43: Number of feature developments started on a given work day (red new features, green bugs fixes, blue ratio of two values; 25-day rolling mean). Data kindly supplied by 7Digital.[1] Github–Local

During the measurement period the number of developers grew from 14 to 35 (the size of its code base and number of customers is not available). The number of developers who worked on each feature was not recorded, and while developers write the software, it is clients who often report most of the bugs (client information is not present in the dataset).

Possible causes for the increase in new feature starts include: an increase in the number of developers, and/or existing developers becoming more skilled at breaking work down into smaller features (i.e., feature implementation time stays about the same because fewer developers are working on each feature, leaving more developers available to start on new features), or having implemented the basic core of the products less effort is now needed to create new features.

A study by Jones and Cullum[929] analysed 8,252 agile tasks whose duration was estimated in hours, with many taking a few hours to complete. Figure 5.44 shows the number of tasks having a given interval between being estimated and starting, and work starting and completing; the lines are fitted regression models (both power laws).

For the 7digital and SiP agile data, knowing that a particular distribution is a good fit is a step towards understanding the processes that generated the measurements (not an end in itself; see section 9.2). More projects need to be analysed to evaluate whether the fitted

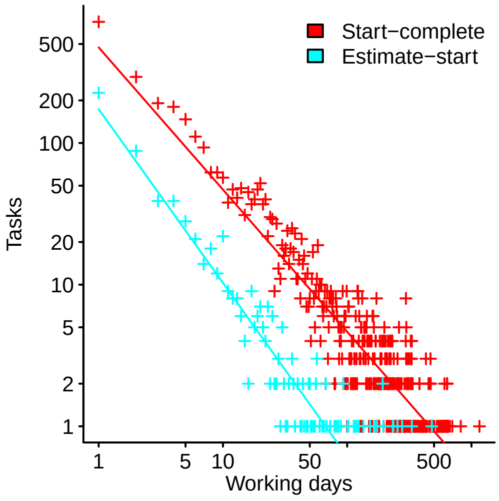


Figure 5.44: Number of tasks having a given duration, in elapsed working days, between estimating/starting (blue), and starting/completing (red). Data from Jones et al.[929] Github–Local

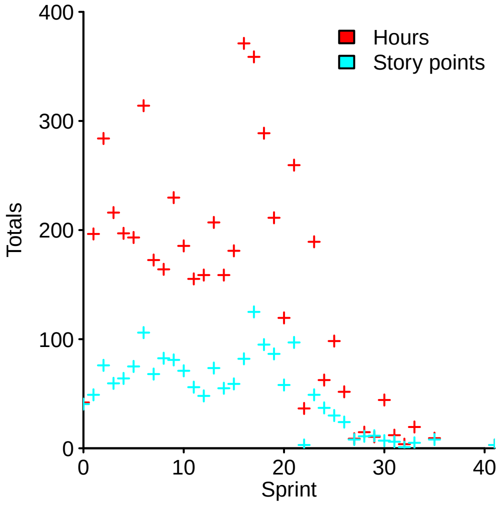


Figure 5.45: Total number of story points and hours worked during each sprint of project P1. Data kindly provided by Vetrò.[1875] Github–Local

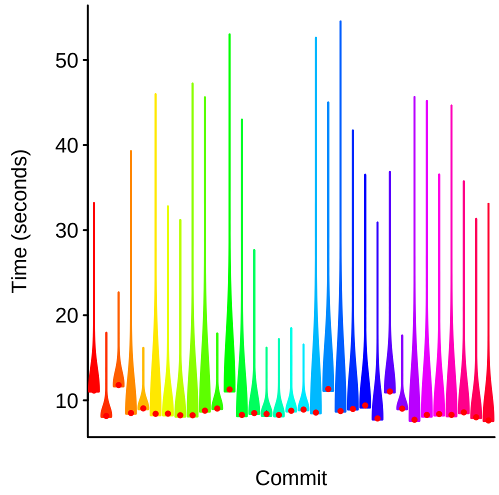


Figure 5.46: Violin plots of benchmark times for a sample of 33 commits to SAX builder (average of 7,357 measurements per commit). Data from Horký.[846] Github–Local

distribution (e.g., Negative Binomial or Power law) is an inconsequential fact, particular to the kind of client interaction, an indicator of inefficient organizational processes, or some other factor.

The Scrum Agile process organizes implementation around a basic unit of time, known as a *sprint*. During a sprint, which has a fixed elapsed time (two weeks is a common choice), the functionality agreed at the start is created; the functionality may be specified using story points and estimated using values from a Fibonacci sequence.

A study by Vetrò, Dürre, Conoscenti, Fernández and Jørgensen[1875] investigated techniques for improving the estimation process for the story points planned for a sprint. Figure 5.45 shows the total number of story points and hours worked during each sprint for project P1 (sprint duration was 2-weeks, up to sprint 26, and then changed to 1-week; 3 to 7 developers worked on the project).

Changes to software sometimes noticeably change its performance characteristics; performance regression testing can be used to check that runtime performance remains acceptable, after source updates. A study by Horký[846] investigated the performance of SAX builder (an XML parser) after various commits to the source tree. Figure 5.46 shows the range of times taken to execute a particular benchmark after a sample of 33 commits (average number of performance measurements per commit was 7,357; red dot is the mean value).

A study by Zou, Zhang, Xia, Holmes and Chen[2012] investigated the use of version control branches during the development of 2,923 projects (which had existed for at least 5-years, had at least 100 commits and three contributors). Figure 5.47 shows the number of projects that have had a given number of branches, with fitted regression line (which excluded the case of a single branch).

5.4.7 Supporting multiple markets

As the amount of functionality supported by a program grows, it may become more profitable to sell targeted configurations (i.e., programs supporting a subset of the available features), rather than one program supporting all features. Reasons for supporting multiple configurations include targeting particular market segments and reducing program resource usage, e.g., the likely cpu time and memory consumed when running the program with the options enabled/disabled.[1682]

Rather than maintaining multiple copies of the source, conditional compilation may be used to select the code included in a particular program build (e.g., using macro names); this topic is discussed in section 7.2.10.

A study by Berger, She, Lotufo, Wąsowski and Czarnecki investigated the interaction between build flags and optional program features in 13 open source projects. Figure 5.48 shows the number of optional features that are enabled when a given number of build flags are set.

5.4.8 Refactoring

Refactoring is an investment that assumes there will be a need to modify the code again in the future, and it is more cost effective to restructure the code now, rather than at some future date. Possible reasons for time shifting an investment in reworking code include: developers not having alternative work to do, or an expectation that the unknown future modifications will need to be made quickly, and it is worth investing time now to reduce future development schedules; also, developers sometimes feel peer pressure to produce source that follows accepted ecosystem norms (e.g., open source that is about to be released).

A justification sometimes given for refactoring is to reduce technical debt. Section 3.2.5 explains why the concept of debt is incorrect in this context, and discusses how to analyse potential investments (such as refactoring).

While models of the evolution of software systems developed with a given cash flow have been proposed,[1567] finding values for the model parameters requires reliable data, which is rarely available.

A study by Kawrykow and Robillard[968] investigated 24,000 change sets from seven long-lived Java programs. They found that between 3% and 16% of all method updates consisted entirely of non-essential modifications, e.g., renaming of local variables, and trivial keyword modifications.

A study by Eshkevari, Arnaoudova, Di Penta, Oliveto, Guéhéuc and Antoniol[551] of identifier renamings in Eclipse-JDT and Tomcat, found that almost half were applied to method names, a quarter to field names, and most of the rest to local variables and parameter names. No common patterns of grammatical form, of the renaming, were found (e.g., changing from a noun to a verb occurred in under 1% of cases). Figure 5.49 shows the number of identifiers renamed in each month, along with release dates; no correlation appears to exist between the number of identifiers renamed and releases.

Other studies[1315, 1866] have found that moving fields and methods, and renaming methods are very common changes.

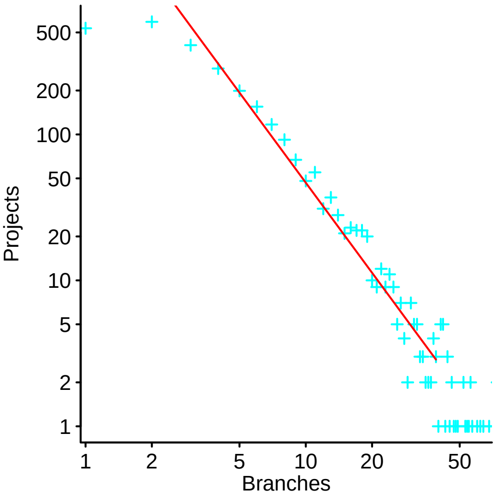


Figure 5.47: Number of projects on Github (out of 2,923) having a given number of branches; the line is a fitted regression model of the form: *projects* ∝ *branches*$^{-2}$. Data from Zou et al.[2012] Github–Local

5.4.9 Documentation

The term *documentation* might be applied to any text: requirements, specifications, code documentation, comments in code, testing procedures, bug reports and user manuals; source code is sometimes referred to as its own documentation or as self-documenting. Section 4.6.4 discusses information sources.

Motivations for creating documentation include:

- a signal of commitment, e.g., management wants people to believe that the project will be around for the long term, or is important,
- fulfil a contract requirement. This requirement may have appeared in other contracts used by the customer, and nobody is willing to remove it; perhaps customer management believes that while they do not understand code, they will understand prose,
- an investment intended to provide a worthwhile return by reducing the time/cost of learning for future project members.

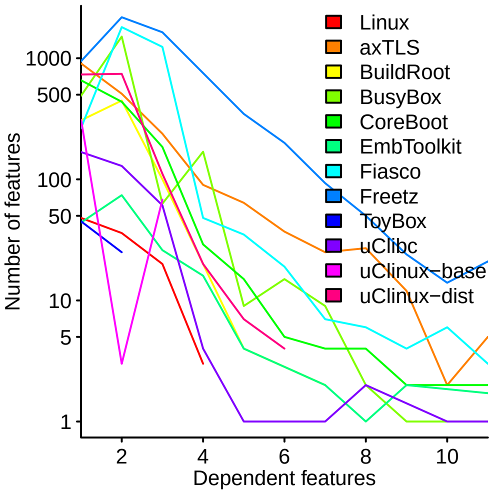


Figure 5.48: Number of optional features selected by a given number of flags. Data kindly provided by Berger.[177] Github–Local

Development projects that derive income from consultancy and training have an incentive to minimise the publicly available documentation. Detailed knowledge of the workings of a software system may be the basis for a saleable service, and availability of documentation reduces this commercial potential.

Issues around interpreting the contents of documentation are covered in chapter 6. The existence of documentation can be a liability, in that courts have considered vendors liable for breach of warranty when differences exist between product behavior and the behavior specified in documentation.[960]

5.4.10 Acceptance

Is the behavior of the software, as currently implemented, good enough for the client to agree to pay for it?

A study by Garman[645] surveyed 70 program and project managers of US Air Force projects about their priorities. Meeting expectations (according to technical specifications) was consistently prioritized higher than being on budget or on time; your author could not model priority decisions by fitting the available explanatory variables using regression; see Github–projects/ADA415138.R.

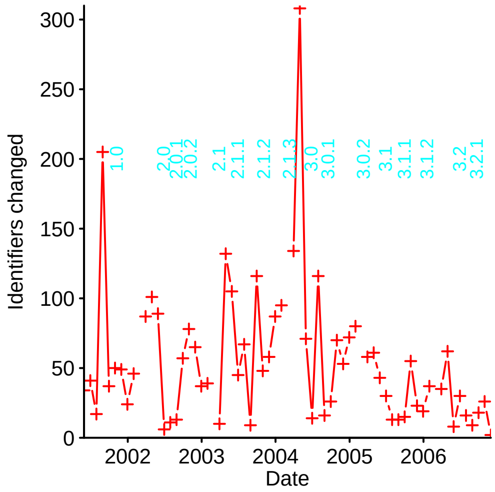


Figure 5.49: Number of identifiers renamed, each month, in the source of Eclipse-JDT; version released on given date shown. Data from Eshkevari et al.[551] Github–Local

Benchmarking of system performance is discussed in section 13.3.

A study by Hofman[831] investigated user and management perceptions of software product quality, and the impact of past quality evaluations on later evaluations; see Github–developers/Hofman-exp1.R.

5.4.11 Deployment

New software systems often go through a series of staged releases (e.g., alpha, followed by beta releases and then release candidates); the intent is to uncover any unexpected customer problems.

Traditionally releases have been either time-based (i.e., specified as occurring on a given date), or feature based (i.e., when a specific set of features has been implemented). Iterative development can enable faster, even continuous, releases (in a rapidly changing market the ability to respond quickly to customer feedback is a competitive advantage issues). Decreasing the friction experienced by a customer, in updating to a new release, increases the likelihood that the release is adopted.

One approach to working towards a release at a future date, is to pause work on new features sometime prior to the release, switching development effort to creating a system that is usable by customers; the term *code freeze* is sometimes used to describe this second phase of development.

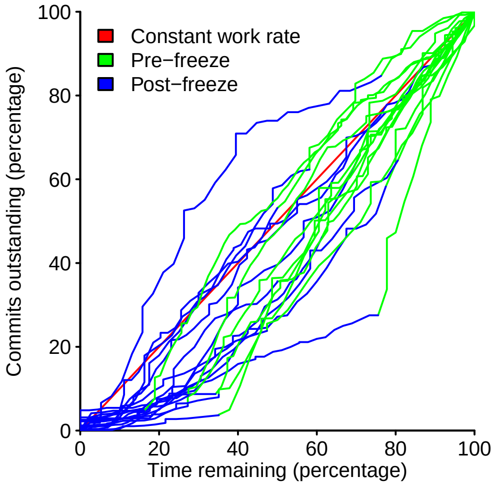


Figure 5.50: Percentage of commits outstanding against percentage the time remaining before deployment, for 18 releases; blue/green transition is the feature freeze date, red line shows a constant commit rate. Data kindly provided by Laukkanen.[1083] Github–Local

A study by Laukkanen, Paasivaara, Itkonen, Lassenius and Arvonen[1083] investigated the number of commits prior to 19 releases of four components of a software system at Ericsson; the length of the code freeze phase varied between releases. Figure 5.50 shows the percentage of commits not yet completed against percentage of time remaining before deployment, for 18 releases; red line shows a constant commit rate, green lines are pre-freeze date, blue lines post-freeze. See Github–projects/laukkanen2017/laukkanen2017.R for component break-down, and regression models.

A basic prerequisite for making a new release is being able to build the software, e.g., being able to compile and link the source to create an executable. Modifications to the source may have introduced mistakes that prevent an executable being built. One study[1831] of 219,395 snapshots (after each commit) of 100 Java systems on Github found that 38% of snapshots could be compiled. Being able to build a snapshot is of interest to researchers investigating the evolution of a system, but may not be of interest to others.

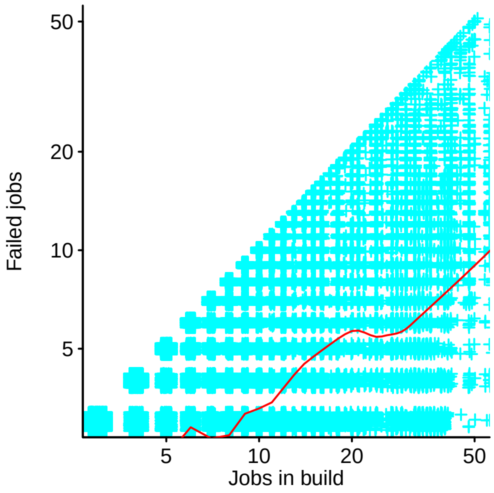


Figure 5.51: Number of failed jobs in Travis CI builds involving a given number of jobs (points have been jittered); line is a loess fit. Data from Gallaba et al.[636] Github–Local

Continuous integration is the name given to the development process where checks are made after every commit to ensure that the system is buildable (regression tests may also be run).

A study by Zhao, Serebrenik, Zhou, Filkov and Vasilescu[1996] investigated the impact of switching project development to use continuous integration (i.e., Travis CI). The regression models built showed that, after the switch, the measured quantity that changed was the rate of increase of monthly merged commits (these slowed considerably, but there was little change in the rate of non-merged commits); see Github–projects/ASE2017.R.

A study by Gallaba, Macho, Pinzger and McIntosh[636] investigated Travis CI logs from 123,149 builds, from 1,276 open source projects; 12% of passing builds contained an actively ignored failure. Figure 5.51 shows the number of failed jobs in each build involving a given number of jobs; line is a loess regression fit.

Building software on platforms other than the one on which it is currently being developed can require a lot of work. One approach, intended to do away with platform specific issues, is virtualization (e.g., Docker containers). One study,[356] from late 2016, was not able to build 34% of a sample of 560 Docker containers available on Github.

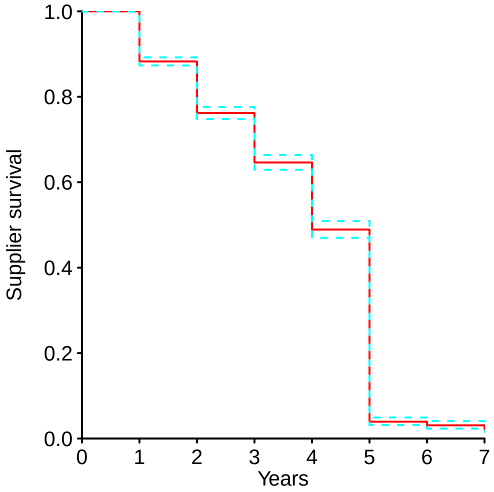


Figure 5.52: Survival curve of IT outsourcing suppliers continuing to work for 2,382 Credit Unions. Data kindly provided by Peukert.[1456] Github–Local

While some bespoke software is targeted at particular computing hardware, long-lasting software may be deployed to a variety of different platforms. The cost/benefit analysis of investing in reducing the cost of porting to a different platform (i.e., reducing the switching cost) requires an estimate of the likelihood that this event will occur.

For systems supporting many build-time configurations options, it may be more cost effective[759] to concentrate on the popular option combinations, and wait until problems with other configurations are reported (rather than invest resources checking many options that will never be used; various configuration sampling algorithms are available[1238]).

A study by Peukert[1456] investigated the switching costs of outsourced IT systems, as experienced by U.S. Credit Unions. Figure 5.52 shows the survival curve of IT outsourcing suppliers employed by 2,382 Credit Unions, over the period 2000 to 2010.

5.5 Development teams

A project needs people in possession of a minimum set of skills (i.e., they need to be capable of doing the technical work), and for individuals to be able to effectively work together as a team. Team members may have to manage a portfolio of project activities.

People working together need to maintain a shared mental model. Manned space exploration is one source for evidence-based studies of team cognition.[463]

Team members might be drawn from existing in-house staff, developers hired for the duration of the project (e.g., contractors), and casual contributors (common for open source projects[92]). Some geographical locations are home to a concentration of expertise in particular application domains (e.g., Cambridge, MA for drug discovery). The O-ring theory[1031] offers an analysis of the benefits, to employers and employees in the same business, of clustering together in a specific location.[617]

If a group of potential team members is available for selection, along with their respective scores in a performance test, it may be possible to select an optimal team based on selecting individuals, but selection of an optimal team is not always guaranteed.[1010] Personnel economics[1087] (i.e., how much people are paid) can also be an important factor in who might be available as a team member.

Developers can choose what company to work for, have some say in what part of a company they work in, and may have control over whether to work with people on the implementation of particular product features.

A study by Bao, Xing, Xia, Lo and Li[127] investigated software development staff turnover within two large companies. A regression model containing the monthly hours worked by individuals fitted around 10% of the variance present in the data; see Github–projects/WhoWillLeaveCompany.R. Figure 5.53 shows the mean number of hours worked per month, plus standard deviation, by staff on two projects (of 1,657 and 834 people).

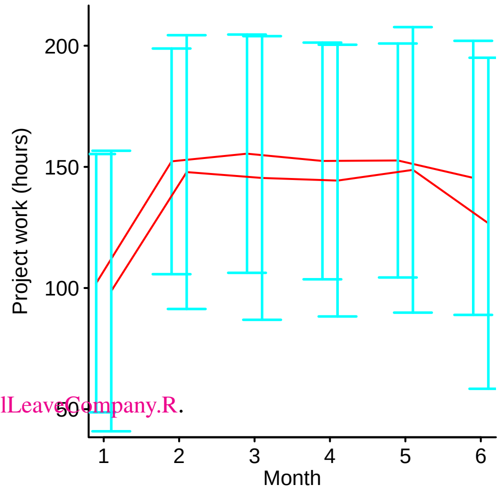


Figure 5.53: Average number of hours worked per month (by an individual), with standard deviation, for two projects staffed by 1,657 and 834 people; two red lines and corresponding error bars offset either side of month value. Data kindly provided by Bao.[127] Github–Local

When a project uses multiple programming languages, a team either needs to include developers familiar with multiple languages, or include additional developers to handle specific languages. Figure 5.54 shows the number of different languages used in a sample of 100,000 GitHub projects (make was not counted as a language).

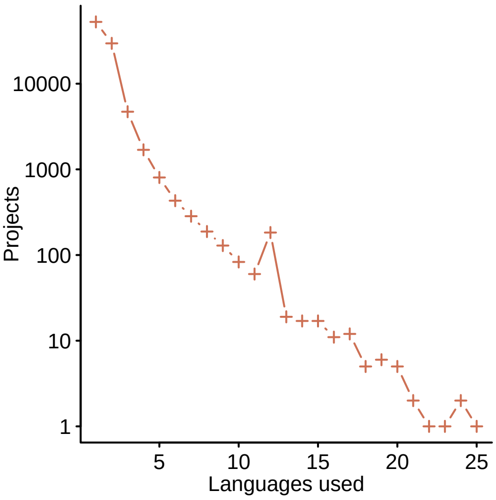


Figure 5.54: Number of projects making use of a given number of different languages in a sample of 100,000 GitHub project. Data kindly provided by Bissyande.[200] Github–Local

What is the distribution of team size for common tasks undertaken on a development project?

Figure 5.55 shows the number of tasks that involved a given number of developers, for tasks usually requiring roughly an hour or two of an individual's time. The SiP tasks are from one company's long-running commercial projects, while the other tasks are aggregated from three companies following the Team Software Process.

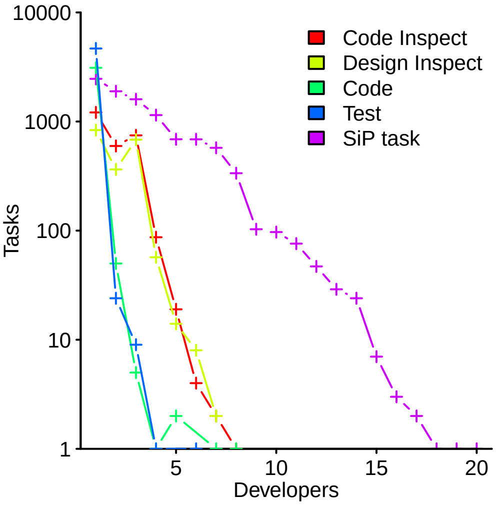


Figure 5.55: Number of tasks worked on by a given number of developers. Data from Nichols et al[1358] and Jones et al.[929] Github–Local

Microsoft's postmortem analysis[896] of the development of what was intended to be Windows office, but became a Windows word processor, illustrates the fits and starts of project development. Figure 5.56 shows the number of days remaining before the planned ship date, for each of the 63 months since the start of the project, against number of full time engineers. The first 12 months were consumed by discussion, with no engineers developing software. See figure 3.17 for a more consistent approach to project staffing.

What is a cost effective way of organizing and staffing a software project team?

There have been few experimental comparisons[1945] of the many project development techniques proposed over the years.

The early software developers were irregulars, in that through trial and error each found a development technique that worked for them. Once software development became a recurring activity within the corporate environment, corporate management techniques were created to try to control the process.[1030]

Drawing a parallel with the methods of production used in manufacturing, the factory concept has been adapted for software projects[418] by several large companies.[x] The claimed advantages of this approach are the same as those it provides to traditional manufacturers, e.g., control of the production process and reduction in the need for highly skilled employees.

The chief programmer team[118] approach to system development was originally intended for environments where many of the available programmers are inexperienced (a common situation in a rapidly growing field); an experienced developer is appointed as the chief programmer, who is responsible for doing detailed development, and allocating the tasks requiring less skill to others. This form of team organization dates from the late 1960s, when programming involved a lot of clerical activity, and in its original formulation emphasis is placed on delegating this clerical activity. Had it been successful, this approach could also be applied to reduce costs in environments where experienced programmers are available (by hiring cheaper, inexperienced programmers).

[x] It was particular popular in Japan.[419] Your author has not been able to locate any data on companies recently using the factory concept to produce software.

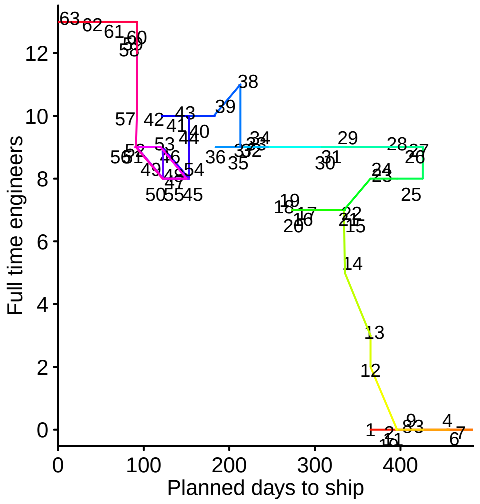


Figure 5.56: Number of days before planned product ship date, against number of full time engineers, for each of the 63 months since the project started (numbers show months since project started). Data from Jackson.[896] Github–Local

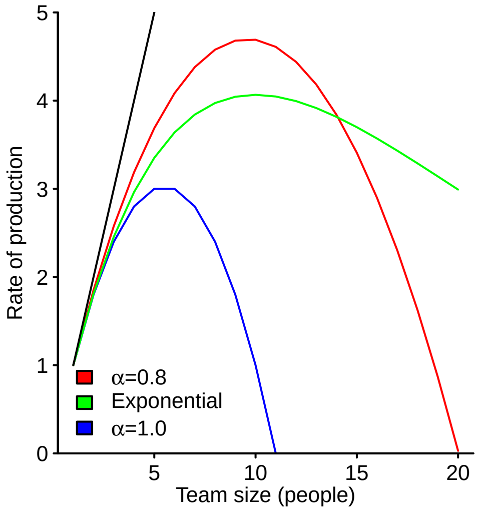


Figure 5.57: Effective rate of production of a team containing a given number of people, with communication overhead $t_0 = t_1 = 0.1$, and various distributions of percentage communication time; black line is zero communications overhead. Github–Local

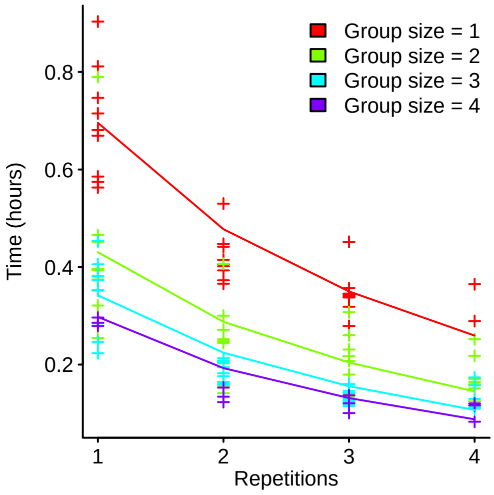


Figure 5.58: Time taken by groups of different sizes to manually assembly a product, over multiple trials; lines are fitted regression models of the form: $Time \propto \frac{0.5-0.2\log(Repetitions)}{Group\_size} - 0.1\log(Repetitions)$. Data kindly provided by Peltokorpi et al.[1444] Github–Local

Once a system has been delivered and continues to be maintained, developers are needed to fix reported problems, and to provide support.[1782] Once initial development is complete, management need to motivate some of those involved to continue to be available to work on the software they are familiar with; adding new features provides a hedonic incentive for existing developers to maintain a connection with the project, and potential new development work is an enticement for potential new team members.

A study by Buettner[269] investigated large software intensive systems, and included an analysis of various staffing related issues, such as: staffing level over time (fig 14.4) and staff work patterns (fig 11.61).

If team member activities are divided into communicating and non-communication (e.g., producing code), how large can a team become before communication activities cause total team output to decline when another person is added?

Assuming that communications overhead,[xi] for each team member, is given by: $t_0(D^\alpha - 1)$, where t_0 is the percentage of one person's time spent communicating in a two-person team, D the number of people in the team and α a constant greater than zero. The peak team size, before adding people starts reducing total output, is given by:[1795]

$$D_{peak} = \left[\frac{1+t_0}{(1+\alpha)t_0}\right]^{\frac{1}{\alpha}}$$

If $\alpha = 1$ (i.e., everybody on the project incurs the same communications overhead), then $D_{peak} = \frac{1+t_0}{2t_0}$, which for small t_0 is: $D_{peak} \approx \frac{1}{2t_0}$. For example, if team members spend 10% of their time communicating with every other team member: $D_{peak} = \frac{1+0.1}{2\times 0.1} \approx 5$.

In this team of five, 50% of each person's time is spent communicating.

If $\alpha = 0.8$, then: $D_{peak} = \left[\frac{1+0.1}{(1+0.8)\times 0.1}\right]^{\frac{1}{0.8}} \approx 10$.

Figure 5.57 shows the effective rate of production (i.e., sum of the non-communications work of all team members) of a team of a given size, whose culture has a particular form of communication overhead.

If people spend most of their time communicating with a few people and very little with the other team members, the distribution of communication time may have an exponential distribution; the (normalised) communications overhead is: $1 - e^{-(D-1)t_1}$, where t_1 is a constant found by fitting data from the two-person team (before any more people are added to the team).

Peak team size is now:

$$D_{peak} = \frac{1}{t_1}, \text{ and, if } t_1 = 0.1, \text{ then: } D_{peak} = \frac{1}{0.1} = 10.$$

In this team of ten, 63% of each persons time is spent communicating (team size can be bigger, but each member will spend more time communicating compared to the linear overhead case).

A study by Peltokorpi and Niemi[1444] investigated the impact of learning and team size on the manual construction of a customised product. Figure 5.58 shows how the time taken to manually assemble the product, for groups containing various numbers of members, decreases with practice. Group learning is discussed in section 3.4.5.

5.5.1 New staff

New people may join a project as part of planned growth, the need to handle new work, existing people leaving, management wanting to reduce the dependency on a few critical people, and many other reasons.

Training people (e.g., developers, documentation writers) who are new to a project reduces the amount of effort available for building the system in the short term. Training is an investment in people, whose benefit is the post-training productivity these people bring to a project.

Brooks' Law[260] says: "Adding manpower to a late software project makes it later", but does not say anything about the impact of not adding manpower to a late project. Under what conditions does adding a person to a project cause it to be delayed?

[xi] This analysis is not backup by any data.

If we assume a new person diverts, from the project they join, a total effort, T_e, in training and that after D_t units of time the trained person contributes E_n effort per unit time until the project deadline; unless the following inequality holds, training a new person results in the project being delayed (in practice a new person's effort contribution ramps up from zero, starting during the training period):

$E_{a1}D_r < (E_{a1}D_t - T_e) + (E_{a2} + E_n)(D_r - D_t)$, where E_{a1} is the total daily effort produced by the team before the addition of a new person, E_{a2} the total daily effort produced by the original team after the addition, and D_r is the number of units of time between the start of training, and the delivery date/time.

Adding a person to a team can both reduce the productivity of the original team (e.g., by increasing the inter-person communication overhead) and increase their productivity (e.g., by providing a skill that enables the whole to be greater than the sum of its parts). Assuming that $E_{a2} = cE_{a1}$, the equation simplifies to: $T_e < (D_r - D_t)(E_n - (1-c)E_{a1})$. If a potential new project member requires an initial investment greater than this value, having them join the team will cause the project deadline to slip.

The effort, T_e, that has to be invested in training a new project member will depend on their existing level of expertise with the application domain, tools being used, coding skills, etc (pretty much everything was new, back in the day, for the project analysed by Brooks, so T_e was probably very high). There is also the important ability, or lack of, to pick things up quickly, i.e., their learning rate.

How often do new staff encounter tasks they have not previously performed?

A study by Jones and Borgatti[928] analysed the tags (having the form @word) used to classify each task in the Renzo Pomodoro dataset; all @words were selected by one person, for the tasks they planned to do each day. Figure 5.59 shows the time-line of the use of @words, with the y-axis ordered by date of first usage. The white lines are three fitted regression models, each over a range of days; the first and last lines have the form: $at\_num = a + b(1 - e^{c \times days})$, and the middle line has the form: $at\_num = a \times days$; with a, b, and c constants fitted by the regression modeling process.

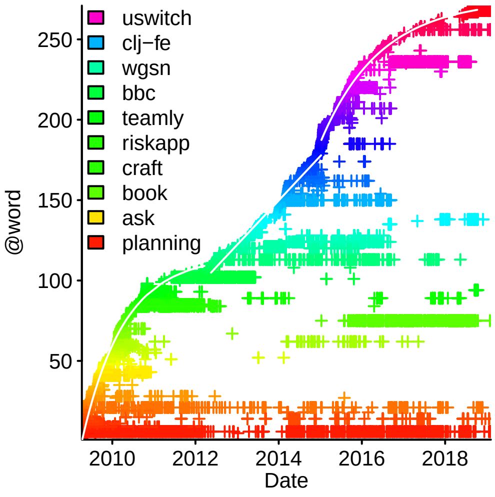


Figure 5.59: Time-line of first @word usage, ordered on y-axis by date of first appearance; legend shows @words with more than 500 occurrences. Data from Jones et al.[928] Github–Local

The decreasing rate of new @words, over two periods (of several years), shows how a worker within a company experienced a declining rate of new tasks, the longer the time spent within a particular role.

5.5.2 Ongoing staffing

Software systems that continue to be used may continue to be supported by software developers, e.g., reported faults addressed and/or new features added.

A study by Elliott[532] investigated the staffing levels for large commercial systems (2 MLOC) over the lifetime of the code (defined as the time between a system containing the code first going live and the same system being replaced; *renewal* was the terminology used by the author); the study did not look at staffing for the writing of new systems or maintenance of existing systems. Figure 5.60 shows the average number of staff needed for renewal of code having a given average lifetime, along with a biexponential regression fit.

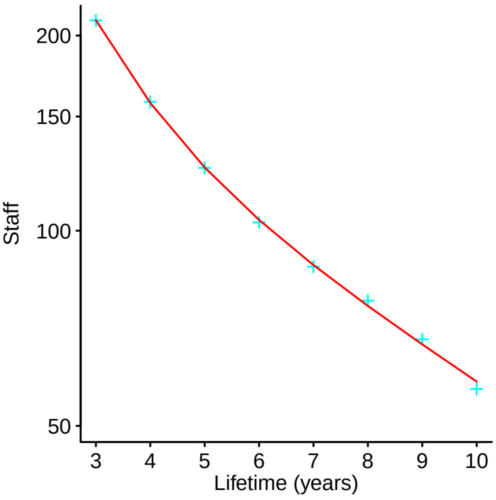


Figure 5.60: Average number of staff required to support renewal of code having a given average lifetime (green); blue/red lines show fitted biexponential regression model. Data extracted from Elliott.[532] Github–Local

A study by Dekleva[471] investigated the average monthly maintenance effort (in hours) spent on products developed using traditional and modern methods (from a 1992 perspective). Figure 5.61 shows the age of systems, and the corresponding time spent on monthly maintenance.

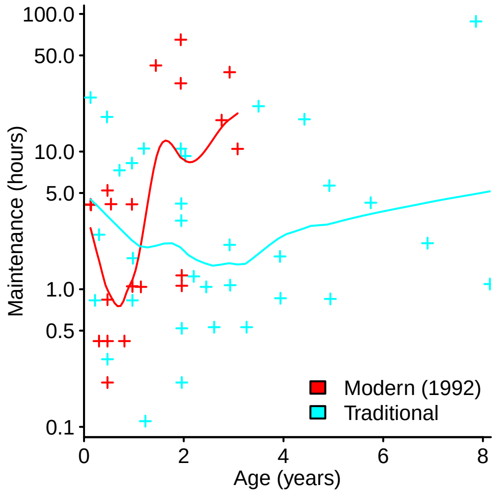


Figure 5.61: Age of systems, developed using one of two methodologies, and corresponding monthly maintenance effort, lines are loess regression fits. Data extracted from Dekleva.[471] Github–Local

5.6 Post-delivery updates

Once operational, software systems are subject to the economics of do nothing, update or replace. The, so-called, maintenance of a software system is a (potentially long term) project in its own right.

Software is maintained in response to customer demand, and changes are motivated by this demand, e.g., very large systems in a relatively stable market[1285] are likely to have a different change profile than smaller systems sold into a rapidly changing market. Reasons motivating vendors to continue to invest in developing commercial products are discussed in chapter 4; also see Github–ecosystems/maint-dev-ratio.R.

Work on software written for academic research projects often stops once project funding drys up. A study[857] of 214 packages associated with papers published between 2001-2015, in the journal Molecular Ecology Resources, found that 73% had not been updated since publication.

Updating software used in safety-critical systems is a non-trivial process;[444] a change impact analysis needs to be made during maintenance, and formal update processes followed.

Buildings are sometimes held up as exemplars of creative items that experience long periods of productive use with little change. In practice buildings that are used, like software, often undergo many changes,[239] and the rearchitecting can be as ugly as that of some software systems. Brand[239] introduced the concept of *shearing layers* to refer to the way buildings contain multiple layers of change (Lim[1131] categorised the RALIC requirements into five layers).

A new release contains code not included in previous releases, and some of the code contained in earlier releases may not be included.

A study by Ozment and Schechter[1414] investigated security vulnerabilities in 15 successive versions of OpenBSD, starting in May 1998. The source added and removed in each release was traced. Figure 5.62 shows the number of lines in each release (x-axis) that were originally added in a given release (colored lines).

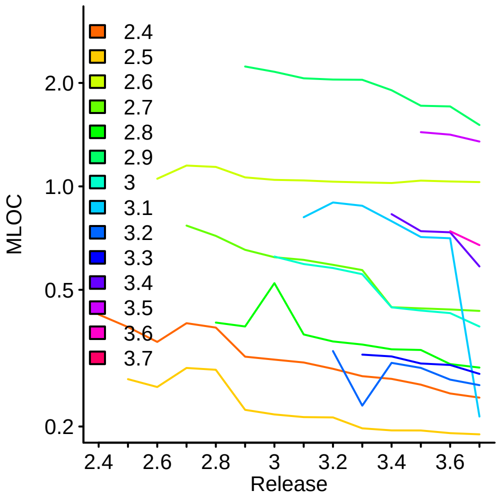


Figure 5.62: Number of lines of code in a release (x-axis) originally added in a given release (colored lines). Data kindly provided by Ozment.[1414] Github–Local

Existing customers are the target market for product updates, and vendors try to keep them happy (to increase the likelihood they will pay for upgrades). Removing, or significantly altering, the behavior of a widely used product feature has the potential to upset many customers (who might choose to stay with the current version that has the behavior they desire). The difficulty of obtaining accurate information on customer interaction with a product incentivizes vendors to play safe, e.g., existing features are rarely removed or significantly changed. If features are added, but rarely removed, a product will grow over time.

Figure 5.63 shows the growth of lines of code, command line options, words in the manual and messages supported by PC-Lint (a C/C++ static analysis tool), in the 11 major releases over 28 years.

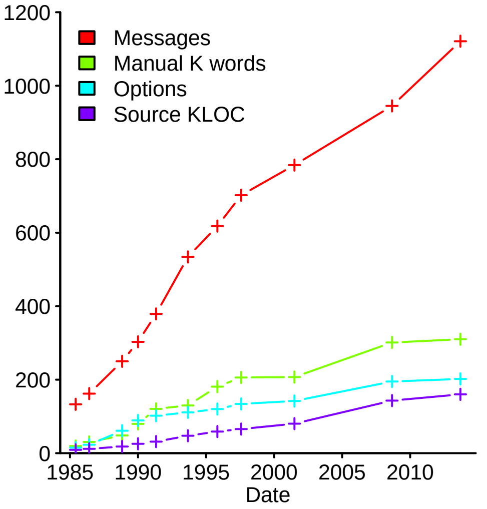


Figure 5.63: Growth of PC-Lint, over 11 major releases in 28 years, of messages supported, command line options, kilo-words in product manual, and thousands of lines of code in the product. Data kindly provided by Gimpel.[674] Github–Local

Companies in the business of providing software systems may be able to charge a monthly fee for support (e.g., fixing problems), or be willing to be paid on an ad-hoc basis.[1762]

An organization using in-house bespoke software may want the software to be adapted, as the world in which it is used changes. The company that has been maintaining software for a customer is in the best position to estimate actual costs, and the price the client is likely to be willing to continue paying for maintenance (see fig 3.23). Without detailed maintenance cost information, another company bidding to take over maintenance of an existing system[195] has to factor in an unknown risk; in some cases the client may be willing to underwrite their risk.

A study by Felici[583] analysed the evolution of requirements, over 22 releases, for eight features contained in the software of a safety-critical avionics system. Figure 5.64 shows the requirements for some features completely changing between releases, while the requirements for other features were unchanged over many releases.

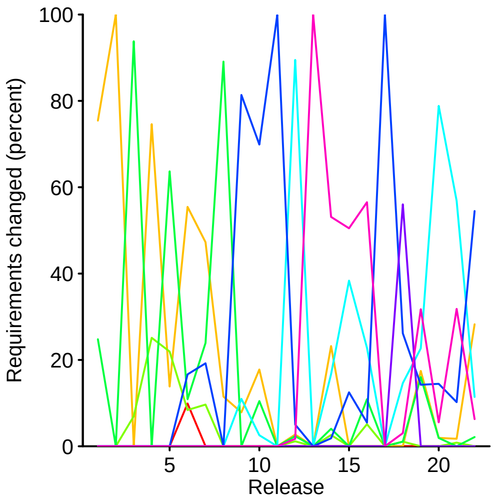


Figure 5.64: Percentage of requirements added/deleted/modified for eight features (colored lines) of a product over 22 releases. Data extracted from Felici.[583] Github–Local

A study by Hatton[779] investigated 1,294 distinct maintenance tasks. For each task, developers estimated the percentage time expected to be spent on adaptive, corrective and perfective activities, this was recorded, along with the actual percentage. Figure 5.65 shows a ternary plot of accumulated (indicated by circle size) estimated and actual percentage time breakdown for all tasks.

With open source projects, changes may be submitted by non-core developers, who do not have access rights to change the source tree.

A study by Baysal, Kononenko, Holmes and Godfrey[150] tracked the progress of 34,535 patches submitted through the WebKit and Mozilla Firefox code review process, between April 2011 and December 2012. Figure 5.66 shows the percentage of patches (as a percentage of submitted patches) being moved between various code review states in WebKit.

Source code is changed via *patches* to existing code. In the case of the Linux kernel submitted patches first have to pass a review process; patches that pass review then have to be accepted by the maintainer of the appropriate subsystem, these maintainers submit patches they consider worth including in the official kernel to Linus Torvalds (who maintains the official version).

A study by Jiang, Adams and German[907] investigated attributes of the patch submission process, such as the time between submission and acceptance (around 30% of the patches that make it through review are accepted into the kernel); the data includes the 81,000+ patches to the kernel source, between 2005 and 2012. Figure 5.67 shows a kernel density plot of the interval between a patch passing review and being accepted by the appropriate subsystem maintainer, and the interval between a maintainer pushing a patch and it being accepted by Torvalds. Maintainers immediately accept half of patches that pass review (Torvalds 6%). The kernel is on roughly an 80 day (sd 12 days) release cycle; the rate at which Torvalds accepts patches steadily increases, before plummeting at the end of the cycle.

Open source projects may be forked, that is a group of developers may decide to take a copy of the code, to work on it as an independent project (see section 4.2.1). Features added to a fork or fixed coding mistakes may be of use to the original project. A study by Zhou, Vasilescu and Kästner[2003] investigated factors that influence whether a pull request submitted to the parent project, by a forked project, are accepted. The 10 variables in the fitted model explained around 25% of the variance[xii]; see Github–economics/fse19-ForkEfficency.R.

A variety of packages implementing commonly occurring application functionality are freely available e.g., database and testing[1986] frameworks.

The need to interoperate with other applications can cause a project to switch the database framework used by an application, or to support multiple database frameworks. The likelihood of an increase/decrease in the number of database frameworks used, and the time spent using different frameworks, is analysed in table 11.6.

5.6.1 Database evolution

Many applications make use of database functionality, with access requests often having the form of SQL queries embedded in strings within the source code. The structure of a database, its schema, may evolve, e.g., columns are added/removed from tables, and tables are added/removed; changes may be needed to support new functionality, or involve a reorganization (e.g., to save space or improve performance).

The kinds of changes made to a schema will be influenced by the need to support existing users (who may not want to upgrade immediately), and the cost of modifying existing code.

A study by Skoulis[1699] investigated changes to the database schema of several projects over time, including: Mediawiki (the software behind Wikipedia and other wikis), and Ensembl (a scientific project). Figure 5.68 shows one database schema in a linear growth phase (like the source code growth seen in some systems, e.g., fig 11.2), while the other has currently stopped growing (e.g., source code example fig 11.52). Systems change in response to customer requirements, and there is no reason to believe that the growth patterns of these two databases won't change.

Figure 5.69 shows the table survival curve for the Mediawiki and Ensembl database schema. Why is the table survival rate for Wikimedia much higher than Ensembl? Perhaps there are more applications making use of the contents of the Wikimedia schema, and the maintainers of the schema don't want to generate discontent in their user-base, or the maintainers are just being overly conservative. Alternatively, uncertainty over what data might be of interest in the Ensembl scientific project may result in the creation of tables that eventually turn out to be unnecessary, and with only two institutions involved, table removal may be an easier decision. The only way of finding out what customer demands are driving the changes is to talk to those involved.

The presence of tables and columns in a schema does not mean the data they denote is used by applications; application developers and the database administrator may be unaware they are unused, or they may have been inserted for use by yet to be written code.

A database may contain multiple tables, with columns in different tables linked using foreign keys. One study[1867] of database evolution found wide variation in the use of

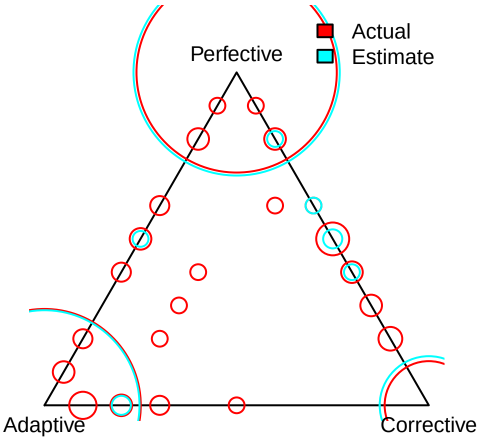


Figure 5.65: Ternary plot showing developers' estimated and actual percentage time breakdown performing adaptive, corrective and perfective work accumulated over 1,294 maintenance tasks; size of accumulation denoted by circle size. Data from Hatton.[779] Github–Local

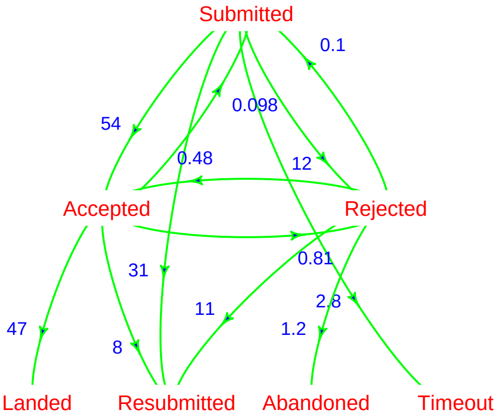


Figure 5.66: Percentage of patches submitted to WebKit (34,535 in total) transitioning between various stages of code review. Data from Baysal et al.[150] Github–Local

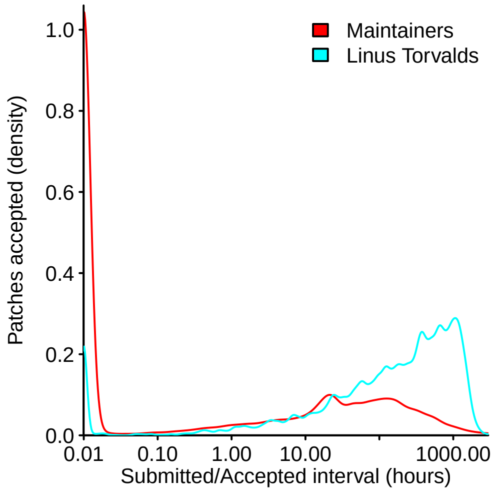


Figure 5.67: Density plot of interval between a patch passing review and being accepted by a maintainer, and interval between a maintainer pushing the patch to Linus Torvalds, and it being accepted into the blessed mainline (only patches accepted by Torvalds included). Data from Jiang et al.[907] Github–Local

[xii] That is, lots of variables performing poorly.

foreign keys, e.g., foreign keys being an integral part of the database, or eventually being completely removed (sometimes driven to a change of database framework).

A study by Blum[210] investigated the evolution of databases and associated programs in the Johns Hopkins Oncology Clinical Information System (OCIS), a system that had been in operational use since the mid-1970s. Many small programs (a total of 6,605 between 1980 and 1988, average length 15 lines) were used to obtain information from the database. Figure 5.70 shows the survival curve for the year of last modification of a program, i.e., the probability that they stopped evolving after a given number of years.

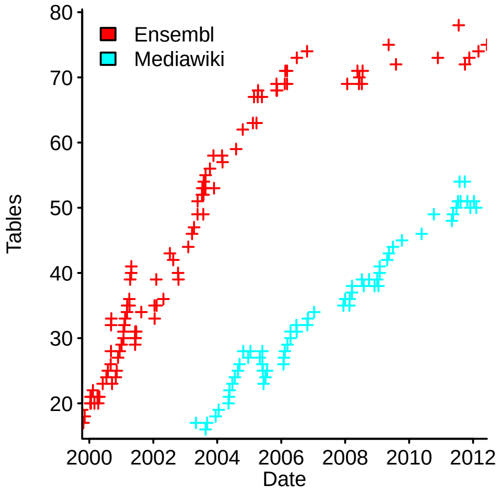


Figure 5.68: Evolution of the number of tables in the Mediawiki and Ensembl project database schema. Data from Skoulis.[1699] Github–Local

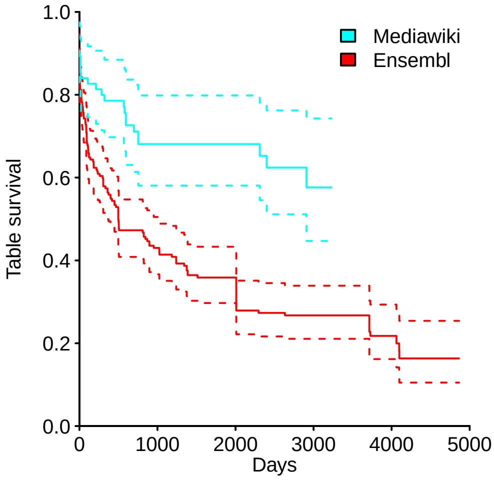


Figure 5.69: Survival curve for tables in Wikimedia and Ensembl database schema, with 95% confidence intervals. Data from Skoulis.[1699] Github–Local

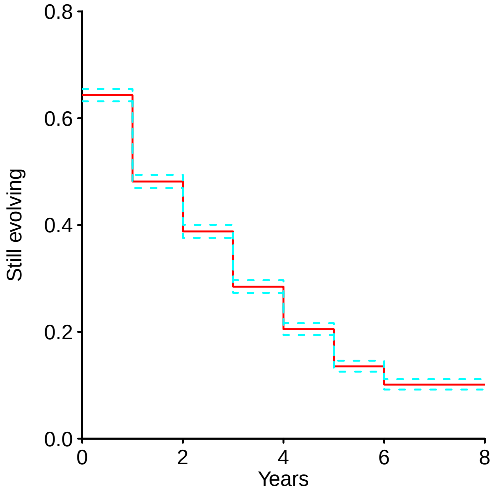


Figure 5.70: Survival curve for year of last modification of database programs, i.e., years before they stopped being changed, with 95% confidence intervals. Data from Blum.[210] Github–Local

Chapter 6

Reliability

6.1 Introduction

People are willing to continue using software containing faults, which they sometimes experience,[i] provided it delivers a worthwhile benefit. The random walk of life can often be nudged to avoid unpleasantness, or the operational usage time can be limited to keep within acceptable safety limits.[254] Regions of acceptability may exist in programs containing many apparently major mistakes, but supporting useful functionality.[1565]

Software systems containing likely fault experiences are shipped because it is not economically worthwhile fixing all of the mistakes made during their implementation; also, finding and fixing mistakes, prior to release, is often constrained by available resources and marketing deadlines.[326]

Software release decisions involve weighing whether the supported functionality provides enough benefit to be attractive to customers (i.e., they will spend money to use it), after factoring in likely costs arising from faults experienced by customers (e.g., from lost sales, dealing with customer complaints and possible fixing reported problems, and making available an updated version).

How many fault experiences will customers tolerate, before they are unwilling to use software, and are some kinds of fault experiences more likely to be tolerated than others (i.e., what is the customer utility function)? Willingness-to-pay is a commonly used measure of risk acceptability, and for safety-critical applications terms such as *As Low As Reasonably Practicable* (ALARP)[757] and *So Far As Is Reasonably Practicable* (SFAIRP) are used.

Some hardware devices have a relatively short lifetime, e.g., mobile phones and graphics cards. Comparing the survival rate of reported faults in Linux device drivers, and other faults in Linux,[1418] finds that for the first 18 months, or so (i.e., 500 days), the expected lifetime of reported fault experiences in device drivers is much shorter than fault experiences in other systems (see figure 6.1); thereafter, the two reported fault lifetimes are roughly the same.

People make mistakes;[1513, 1545] economic considerations dictate how much is invested in reducing the probability that mistakes leading to costly fault experiences remain (either contained in delivered software systems, or as a component of a larger system). The fact that programs often contained many mistakes was a surprise to the early computer developers,[1940] as it is for people new to programming.[ii]

Developers make the coding mistakes that create potential fault experiences, and the environment in which the code executes provides the input that results in faults occurring (which may be experienced by the user). This chapter discusses the kinds of mistakes made, where they occur in the development process, methods used to locate them and techniques for estimating how many fault experiences can potentially occur. Issues around the selection of algorithms is outside the scope of this chapter; algorithmic reliability issues include accuracy[474] and stability of numerical algorithms,[817] and solutions include minimising the error in a dot product by normalizing the values being multiplied.[533]

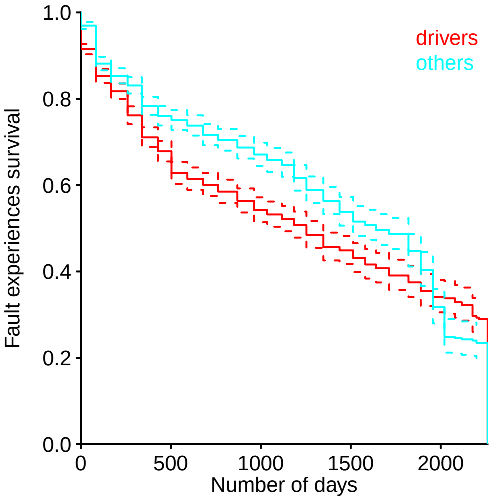


Figure 6.1: Survival rate of reported fault experiences in Linux device drivers and the other Linux subsystems. Data from Palix et al.[1418] Github–Local

[i] Experience is the operative word, a fault may occur and not be recognized as such.

[ii] The use of assertions for checking program behavior was proposed by Turing in 1949,[1295] and was later reinvented by others.

Operating as an engineering discipline does not in itself ensure that a constructed system has the desired level of reliability. There has been, roughly, a 30-year cycle for bridge failures;[1455] new design techniques and materials are introduced, and growing confidence in their use leads to overstepping the limits of what can safely be constructed.

What constitutes reliability, in a given context, is driven by customer requirements, e.g., in some situations it may be more desirable to produce an inaccurate answer than no answer at all, while in other situations no answer is more desirable than an inaccurate one.

Program inputs that cause excessive resource usage can also be a reliability issue. Examples of so-called *denial of service* attacks include regular expressions that are susceptible to nonlinear, or exponential, matching times for certain inputs.[435]

The early computers were very expensive to buy and operate, and much of the software written in the 1960s and 1970s was funded by large corporations or government bodies; the US Department of Defence took an active role in researching software reliability, and much of the early published research is based on the kinds of software development projects undertaken on behalf of the DOD and NASA projects during this period.

The approach to software system reliability promoted by these early research sponsors set the agenda for much of what has followed, i.e., a focus on large projects that are expected to be maintained over many years, or systems operating in situations where the cost of failure is extremely high and there is very limited time, if any, to fix issues (e.g., Space shuttle missions[698]).

This chapter discusses reliability from a cost/benefit perspective; the reason that much of the discussion involves large software systems is that a data driven discussion has to follow the data, and the prexisting research focus has resulted in more data being available for large systems. Mistakes have a cost, but these may be outweighed by the benefits of releasing the software containing them. As with the other chapters, the target audience is software developers and vendors, not users; it is possible for vendors to consider a software system to be reliable because it has the intended behavior, but for many users to consider it unreliable because it does not meet their needs.

The relative low cost of modifying existing software, compared to hardware, provides greater flexibility for trading-off upfront costs against the cost of making changes later (e.g., by reducing the amount of testing before release), knowing that it is often practical to provide updates later. For some vendors, the Internet provides an almost zero cost update distribution channel.

In some ecosystems it is impractical or impossible to update software once it has been released, e.g., executable code associated with the Ethereum cryptocurrency is stored on a blockchain (coding mistakes can have permanent crippling consequences[1807]).

Mistakes in software can have a practical benefit for some people, for instance, authors of computer malware have used mistakes in cpu emulators to detect that their activity may be monitored[1416] (and therefore the malware should remain inactive).

Mistakes are not unique to software systems; a study[1252] of citations in research papers found an average error rate of 20%.

Proposals[826] that programming should strive to be more like mathematics are based on the misconception that the process of creating proofs in mathematics is less error prone than creating software.[446]

The creation of mathematics shares many similarities with the creation of software, and many mistakes are made in mathematics;[1491] mathematical notation is a language with rules specifying syntax and permissible transformations. The size, complexity and technicality of modern mathematical proofs has raised questions about the ability of anybody to check whether they are correct, e.g., Mochizuki's proof of the *abc* conjecture,[299] and the Hales-Ferguson proof of the Kepler Conjecture.[1056] Many important theorems don't have proofs, only sketches of proofs and outline arguments that are believed to be correct;[1336] the sketches provide evidence used by other mathematicians to decide whether they believe a theorem is true (a theorem may be true, even though mistakes are made in the claimed proofs).

Mathematical proof differs from software in that the proof of a theorem may contains mistakes, and the theorem may still be true. For instance, in 1899 Hilbert found mistakes[1907] in Euclid's Elements (published around 300 BC); the theorems were true, and Hilbert was able to add the material needed to correct the proofs. Once a theorem is believed to be true, mathematicians have no reason to check its proof.

The social processes involved in the mathematics community coming to believe that a theorem is true, is evolving, to come to terms with believing machine-checked proofs.[1482] The nature and role of proof in mathematics continues to be debated.[810]

Mistakes are much less likely to be found in mathematical proofs than software, because a lot of specialist knowledge is needed to check new theorems in specialised areas, but a clueless button pusher can experience a fault in software simply by running it; also, there are few people checking proofs, while software is being checked every time it is executed. One study[305] found that 7% of small random modifications to existing proofs, written in Coq, did not cause the proof to be flagged as invalid.

Fixing a reported fault experience is one step in a chain of events that may result in users of the software receiving an update.

For instance, operating system vendors have different approaches to the control they exercise over the updates made available to customers. Apple maintains a tight grip over use of iOS, and directly supplies updates to customers cryptographically signed for a particular device (i.e., the software can only be installed on the device that downloaded it). Google supplies the latest version of Android to OEMs and has no control over what, if any, updates these OEMs supply to customers (who may chose to install versions from third-party suppliers). Microsoft sells Windows 10 through OEMs, but makes available security fixes and updates for direct download by customers.

Figure 6.2 shows some of the connections between participants in the Android ecosystem (number of each kind in brackets), and some edges are labeled with the number of known updates flowing between particular participants (from July 2011 to March 2016).

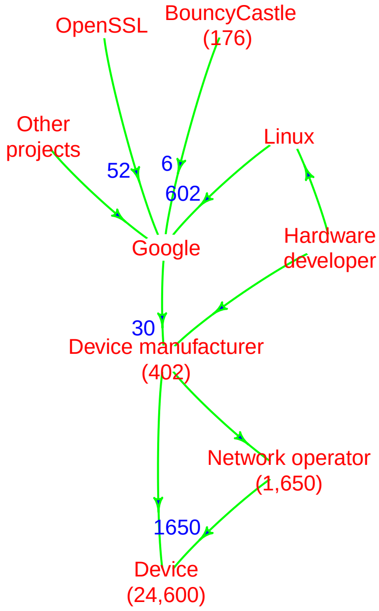


Figure 6.2: Flow of updates between participants in one Android ecosystem; number of each kind of member given in brackets, number of updates shipped on edges (in blue). Data from Thomas.[1809] Github–Local

Experiments designed to uncover unreliability issues may fail to find any. This does not mean that they are rare, the reason for failing to find a mistake may be lack of statistical power (i.e., the likelihood of finding an effect if one exists); this topic is discussed in section 10.2.3.

The software used for some applications is required to meet minimum levels of reliability, and government regulators (e.g., Federal Aviation Administration) may be involved in some form of certification (these regulators may not always have the necessary expertise, and delegate the work to the vendor building the system[501]).

In the U.S., federal agencies are required to adhere to an Executive Order[iii] that specifies: “Regulatory action shall not be undertaken unless the potential benefits to society for the regulation outweigh the potential costs to society.” In some cases the courts have required that environmental, social and moral factors be included in the cost equation.[303]

Quality control: Manufacturing hardware involves making a good enough copy of a reference product. Over time[375] manufacturers have developed quality control techniques that support the consistent repetition of a production process, to deliver a low defect finished product. Software manufacturing involves copying patterns of bits, and perfect copies are easily made. The quality assurance techniques designed for the manufacture hardware are solving a problem relevant to software production.

The adoption of a quality certification process by an organization, such as ISO 9000, may be primarilysymbolic.[805]

6.1.1 It’s not a fault, it’s a feature

The classification of program behavior as a fault, or a feature, can depend on the person doing the classification (e.g., user or developer). For instance, software written to manage a parts inventory may not be able to add a new item once the number of items it contains equals 65,536; a feature/fault that users will not encounter until an attempt is made to add an item that would take the number of parts in the inventory past this value.

Studies[486, 811] of fault reports have found that many of the issues are actually requests for enhancement.

The choice of representation for numeric values places maximum/minimum bounds on the values that can be represented, and in the case of floating-point a percentage granularity on representable values. Business users need calculations on decimal values to be exact, something that is not possible using binary floating-point representations (unless emulated in software), and business oriented computers sometimes include hardware support

[iii] President Reagan signed Executive Order 12291 in 1981, and subsequent presidents have issued Executive Orders essentially affirming this requirement.[1776]

for decimal floating-point operations; in other markets, implementation costs[407] resulted in binary becoming the dominant hardware representation for floating-point. While implementation costs eventually decreased to a point where it became commercially viable for processors to support both binary and decimal, much existing software has been written for processors that use a binary floating-point representation.

Intel's Pentium processor was introduced in 1993, as the latest member of the x86 family of processors. Internally the processor contains a 66-bit hardwired value for π.[iv] A double precision floating-point value is represented using a 53-bit mantissa, which means internal operations involving values close to π (e.g., using one of the instructions that calculate a trigonometric function) may have only 13-bits of accuracy (i.e., $66-53$). To ensure that the behavior of new x86 family processors are consistent with existing processors, subsequent processors have continued to represented π internally using 66-bits (rather than the 128-bits needed to achieve an accuracy of better than 1.5 ULP).

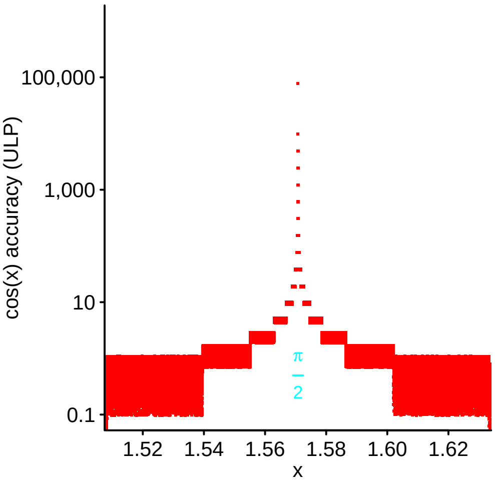


Figure 6.3: Accuracy of the value returned by the cos instruction on an Intel Core i7, for 52,521 argument values close to $\frac{\pi}{2}$. Data kindly provided by Duplichan.[1642] Github–Local

Figure 6.3 shows the error in the values returned by the cos instruction in Intel's Core i7-2600 processor, for 52,521 argument values close to $\frac{\pi}{2}$, expressed in units in the last place (ULP); from a study by Duplichan.[1642]

The variation in the behavior of software between different releases, or running the same code on different hardware can be as large as the behavior affect the user is looking for, potentially a serious issue when medical diagnosis is involved.[738]

The accuracy of calculated results may be specified in the requirements, or the developer writing the code may be the only person to give any thought to the issue. Calculations involving floating-point values may not be exact, and the algorithms used may be sensitive to rounding errors.[1942] Developers may believe they are playing safe by using variables declared to have types capable of representing greater accuracy than is required[1597] (leading to higher than necessary resource usage,[718] e.g., memory, time and battery power). Small changes to numerical code can produce a large difference in the output produced,[1793] and simple calculations may require complex code to correctly implement, e.g., calculating $\sqrt{a^2+b^2}$.[218]

6.1.2 Why do fault experiences occur?

Two events are required for a running program to experience a software related fault:

- a mistake exists in the software,
- the program processes input values that cause it to execute the code containing the mistake in a way that results in a fault being experienced[404] (software that is never used has no reported faults).

Some coding mistakes are more likely to be encountered than others, because the input values needed for them to trigger a fault experience are more likely to occur during the use of the software. Any analysis of software reliability has to consider the interplay between the probabilistic nature of the input distribution, and coding mistakes present in the source code (or configuration information).

An increase in the number of people using a program is likely to lead to an increase in fault reports, because of both an increase in possible reporters and an increase in the diversity of input values.

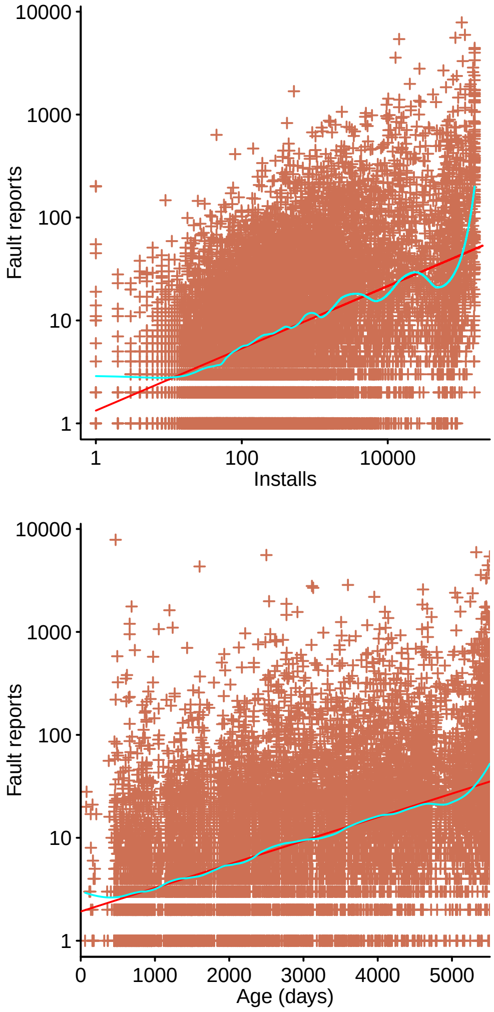


Figure 6.4: Reported faults against number of installations (upper) and age (lower). Data from the "wheezy" Debian release.[1842] Github–Local

The Ultimate Debian Database project[1842] collects information about packages included in the Debian Linux distribution, from users who have opted-in to the Debian Popularity Contest. Figure 6.4 shows the numbers of installs (for the "wheezy" release) of each packaged application against faults reported in that package, and also age of the package against faults reported (data from the Debian Bug Tracking System, which is not the primary fault reporting system for some packages); also, see fig 11.23. A fitted regression model is:

$$reported\_bugs = e^{-0.15+0.17\log(insts)+(30+2.3\log(insts))\times age\times 10^{-5}}$$

For an *age* between 1,000–6,000 and installs between 10–20,000 ($\log(insts)$ is between 2–10), the number of installations (a proxy for number of users) appears to play a larger role in the number of reported faults, compared to *age* (i.e., the amount of time the package has been included in the Debian distribution). The huge amount of variance in the data points to other factors having a major impact on number of reported faults.

[iv] The 66-bit value is: `C90FDAA2 2168C234 C`, while the value to 192-bits is: `C90FDAA2 2168C234 C4C66 28B 80DC1CD1 29024E08 8A67CC74`.

A study[1765] of TCP checksum performance found that far fewer corrupt network packets were detected in practice, than expected (by a factor of between 10 and 100). The difference between practice and expectation was found to be caused by the non-uniform distribution of input values (the proof that checksum values are uniformly distributed assumes a uniform distribution of input values).

A hardware fault may cause the behavior of otherwise correctly behaving software to appear to be wrong; hardware is more likely to fail as the workload increases.[892]

6.1.3 Fault report data

While fault reports have been studied almost since the start of software development, until open source bug repositories became available there was little publicly available fault report data. The possibility of adverse publicity, and the fear of legal consequences from publishing information on mistakes found in software products was not lost on commercial organizations, with nearly all of them treating such information as commercially confidential. While some companies maintained software fault report databases,[716] these were rarely publicly available.

During the 1970s, the Rome Air Defence Center published many detailed studies of software development,[417] and some included data on faults experienced by military software projects.[1947] However, these reports were not widely known about, or easy to obtain, until they became available via the Internet; a few studies were published as books.[1806]

The few pre-Open source datasets analysed in research papers contained relatively small numbers of fault reports, and if submitted for publication today would probably be rejected as not worthy of consideration. These studies[1451, 1453] usually investigated particular systems, listing percentages of faults found by development phase, and the kinds of faults found; one study[1898] listed fault information relating to one application domain: medical device software. An early comprehensive list of all known mistakes in a widely used program was for LaTeX.[1018]

The economic impact of loss of data, due to poor computer security, has resulted in some coding mistakes in some programs (e.g., those occurring in widely used applications that have the potential to allow third parties to gain control of a computer) being recorded in security threat databases. Several databases of security related issues are actively maintained, including: the NVD (National Vulnerability Database[1366]), the VERIS Community Database (VCDB);[1868] an Open source vulnerability database, the Exploit database (EDB)[521] lists proof of concept vulnerabilities; mistakes in code that may be exploited to gain unauthorised access to a computer (vulnerabilities discovered by security researchers who have a motivation to show off their skills), and the Wooyun program.[1994]

While many coding mistakes exist in widely used applications, only a tiny fraction are ever exploited to effectively mount a malicious attack on a system.[1450]

In some application domains the data needed to check the accuracy of a program's output may not be available, or collected for many years, e.g., long range weather forecasts. The developers of the Community Earth System Model compare the results from previous climate calculations to determine whether modified code produces statistically different results.[117]

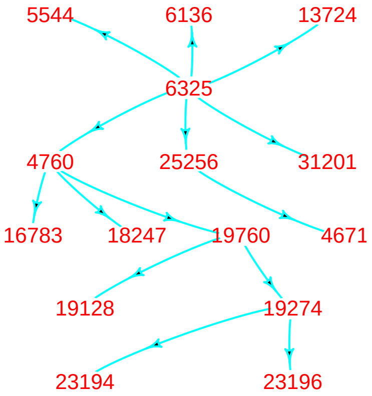


Figure 6.5: Duplicates of Eclipse fault report 4671 (report 6325 was finally chosen as the master report); arrows point to report marked as duplicate of an earlier report. Data from Sadat et al.[1603] Github–Local

A study by Sadat, Bener and Miranskyy[1603] investigated issues involving connecting duplicate fault reports (i.e., reports involving the same mistake). Figure 6.5 shows the connection graph for Eclipse report 6325 (report 4671 was the earliest report covering this issue).

Fixing a mistake may introduce a new mistake, or may only be a partial fix, i.e., fixing commits may be a continuation of a fix for an earlier fault report.

A study by Xiao, Zheng, Jiang and Sui[1965] investigated *regression* faults in Linux (the name given to fault experiences involving features that worked correctly, up until a code change). Figure 6.6 shows a graph of six fault reports for Linux (in red), the commits believed to fix the coding mistake (in blue), and subsequent commits needed to fix mistake(s) introduced by an earlier commit (in blue, follow arrows).

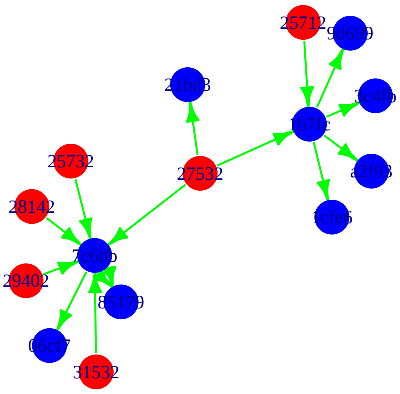


Figure 6.6: Six fault reports (red), their associated bug fixing commits (blue), and subsequent commits to fix mistakes introduced by the earlier commit (blue). Data from Xiao et al.[1965] Github–Local

How similar are the characteristics of Open source project fault report data, compared to commercial fault report data?

Various problems have been found with Open source fault report data, which is not to say that fault report data on closed source projects is not without its own issues; known problems include:

- reported faults do not always appear in the fault report databases (e.g., serious bugs tend to be under-reported in commit logs[197, 1353]). One study[101] of fault reports for Apache, over a 6-week period, found that only 48% of bug fixes were recorded as faults in the Bugzilla database; the working practice of the core developers was to discuss serious problems on the mailing list, and many fault experiences were never formally logged with Bugzilla,

- fault reports are misclassified. One study of fault reports[811] (also see section 14.1.1) found that 42.6% of fault reports had been misclassified, with 39% of files marked as defective not actually containing any reported fault (e.g., were requests for enhancement),

- reporting bias: fault experiences discovered through actively searching for them,[566] rather than normal program usage (e.g., Linux is a popular target for researchers using fuzzing tools, and `csmith` generated source code designed to test compilers). The reporting of vulnerabilities contained, or not, in the NVD has been found to be driven by a wide variety social, technical and economic pressures.[351, 1354] Data on fault reports discovered through an active search process may have different characteristics compared to faults experienced through normal program usage,

- the coding mistake is not contained within the source of the program cited, but is contained within a third-party library. One study[1165] surveyed developers about this issue,

- fault experience could not be reproduced or was intermittent: a study[1606] of six servers found that on average 81% of the 266 fault reports analysed could be reproduced deterministically, 8% non-deterministically, 9% were timing dependent, plus various other cases,

Fault report data does not always contain enough information to answer the questions being asked of it, e.g., using incidence data to distinguish between different exponential order fault growth models[1266] (information is required on the number of times the same fault experience has occurred, see section 4.3.2).

6.1.4 Cultural outlook

Cultures vary in their members' attitude to the risk of personal injury and death; different languages associate different concepts with the word *reliability* (e.g., Japanese[1272]), and the English use of the term *at risk* has changed over time.[2009] A study by Viscusi and Aldy[1885] investigated the value of a statistical life in 11 countries, and found a range of estimates from $0.7 million to $20 million (adjusted to the dollar rate in 2000). Individuals in turn have their own perception of risk, and sensitivity to the value of life.[1707] Some risks may be sufficiently outside a persons' experience that they are unable to accurately estimate their relative likelihood,[1122] or be willing to discount an occurrence affecting them, e.g., destruction of a city, country, or all human life, by a meteor impact.[1715]

Public perception of events influences, and is influenced by, media coverage (e.g., in a volcano vs. drought disaster, the drought needs to kill 40,000 times as many people as the volcano to achieve the same probability of media coverage[528]). A study[528] of disasters and media coverage found that when a disaster occurs at a time when other stories are deemed more newsworthy, aid from U.S. disaster relief is less likely to occur.

Table 6.1, from a 2011 analysis by the UK Department for Transport,[1821] lists the average value that would have been saved, per casualty, had an accident not occurred. A report[346] from the UK's Department for Environment, Food and Rural Affairs provides an example of the kind of detailed analysis involved in calculating a monetary valuation for reducing risk.

Injury severity	**Lost output**	**Human costs**	**Medical and ambulance**	**Total**
Fatal	£545,040	£1,039,530	£940	£1,585,510
Serious	£21,000	£144,450	£12,720	£178,160
Slight	£2,220	£10,570	£940	£13,740
Average	£9,740	£35,740	£2,250	£47,470

Table 6.1: Average value of prevention per casualty, by severity and element of cost (human cost based on willingness-to-pay values); last line is average over all casualties. Data from UK Department for Transport.[1821]

A study by Costa and Kahn[401] investigated changes in the value of life in the USA, between 1940 and 1980. The range of estimates, adjusted to 1990 dollars, was $713,000 to $996,000 in 1940, and $4.144 million to $5.347 million in 1980.

Some government related organizations, and industrial consortia, have published guidelines covering the use of software in various applications, e.g., in medical devices,[573] cybersecurity,[1537] and automotive.[89, 1274, 1275] Estimates of the number of deaths associated with computer related accidents contain a wide margin of error.[1173]

Governments are aware of the dangers of society becoming overly risk-averse, and some have published risk management policies.[1896]

People often express uncertainty using particular phrases, rather than numeric values. Studies[1299] have found that when asked to quantify a probabilistic expression, the range of values given can be quite wide.

A study by Budescu, Por, Broomell and Smithson[268] investigated how people in 24 countries, speaking 17 languages, interpreted uncertainty statements containing four probability terms, i.e., very unlikely, unlikely, likely and very likely, translated to the subjects' language. Figure 6.7 shows the mean percentage likelihood estimated by people in each country to statements containing each term.

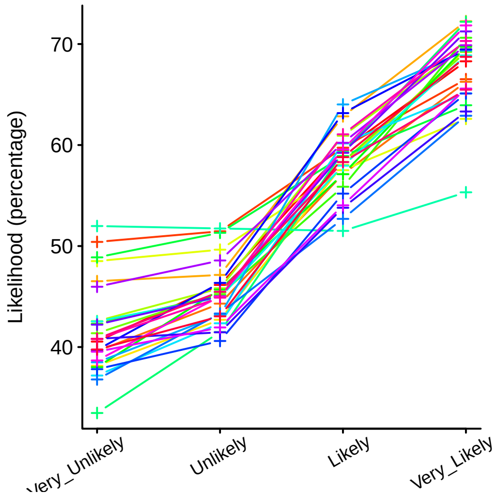


Figure 6.7: Mean percentage likelihood of (translated) statements containing a probabilistic term; one coloured line per country. Data from Budescu et al.[268] Github–Local

The U.S. Department of Defense Standard MIL-STD-882E[476] defines numeric ranges for some words that can be used to express uncertainty, when applied to an individual item; these words include:

- *Probable*: "Will occur several times in the life of an item"; probability of occurrence less than 10^{-1} but greater than 10^{-2}.
- *Remote*: "Unlikely, but possible to occur in the life of an item"; probability of occurrence less than 10^{-3} but greater than 10^{-6}.
- *Improbable*: "So unlikely, it can be assumed occurrence may not be experienced in the life of an item"; probability of occurrence less than 10^{-6}.

Some phrases are used to express relative position on a scale, e.g., hot/warm/cold water describe position on a temperature scale. A study by Sharp, Paul, Nagesh, Bell and Surdeanu[1658] investigated, so-called *gradable adjectives* (e.g., huge, small). Subjects saw statements such as: “Most groups contain 1470 to 2770 mards. A particular group has 2120 mards. There is a moderate increase in this group.”; subjects were then asked: “How many mards are there?”.

Figure 6.8 shows violin plots for the responses given to statements/questions involving various gradable quantity adjectives (see y-axis); the x-axis is in units of standard deviation from the mean response, i.e., a normalised scale.

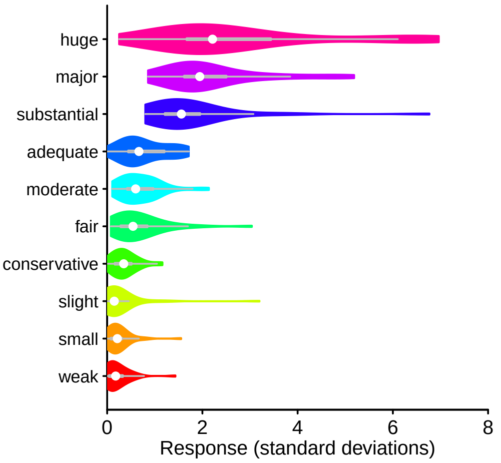


Figure 6.8: Subjects' perceived change in the magnitude of a quantity, when the given gradable size adjective is present. Data from Sharp et al.[1658] Github–Local

The interpretation of a quantifier (i.e., a word indicating quantity) may be context dependent. For instance, “few of the juniors were accepted” may be interpreted as: a small number were accepted; or, as: less than the expected number were accepted.[1845]

6.2 Maximizing ROI

A vendor's approach to product reliability is driven by the desire to maximize return on investment. The reliability tradeoff involves deciding how much to invest in finding and fixing implementation mistakes prior to release, against fixing fault experiences reported after release. Factors that may be part of the tradeoff calculation include:

- some mistakes will not generate fault experiences, and it is a waste of resources finding and fixing them. The lack of fault experiences, for a particular coding mistake, may be a consequence of software having a finite lifetime (see fig 3.7), or the source code containing the mistake being rewritten before it causes a fault to be experienced.

  A study by Di Penta, Cerulo and Aversano[485] investigated the issues reported by three static analysis tools (Rats, Splint and Pixy), when run on each release of several large software systems (e.g., Samba, Squid and Horde). The reported issues were processed, to find the first/last release where each issue was reported for a particular line of code (in some cases the issue was reported in the latest release).

  Figure 6.9 shows the survival curve for the two most common warnings reported by Splint (memory problem and type mismatch, make up over 85% of all generated warnings), where the warnings ceased because of code modifications that were not the result of a reported fault being fixed; also see fig 11.80.

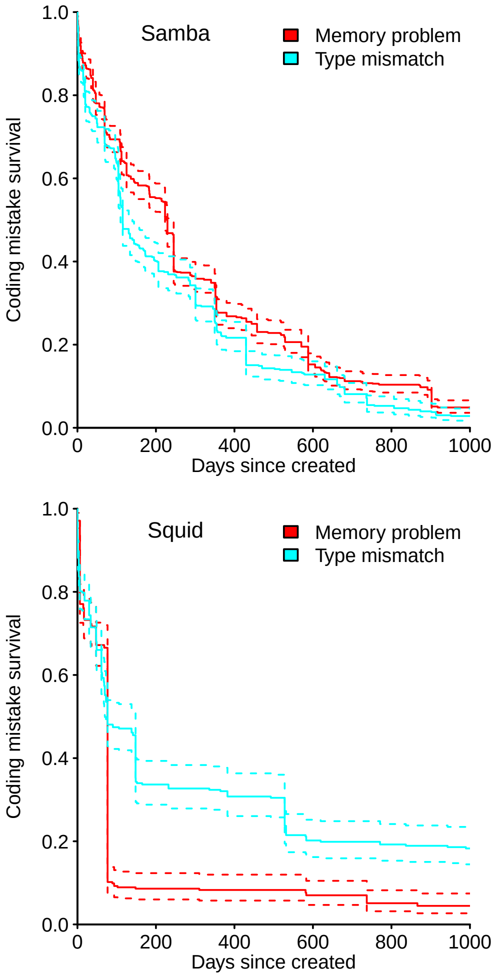


Figure 6.9: Survival curves of the two most common warnings reported by Splint in Samba and Squid, where survival was driven by code changes and not fixing a reported fault; with 95% confidence intervals. Data from De Penta et al.[485] Github–Local

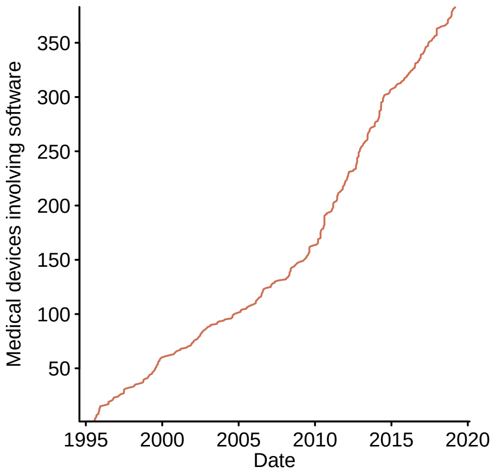


Figure 6.10: Cumulative number of class III (high-risk) medical devices, containing software in their product summary, achieving premarket approval from the FDA. Data from FDA.[574] Github–Local

The average lifetime of coding mistakes varies between programs and kind of mistake.[1417]

- there may be a competitive advantage to being first to market with a new or updated product; the largest potential costs may be lost sales, rather than the cost of later correction of reported faults, i.e., it may be worth taking the risk that customers are not deterred by the greater number of fault experiences; so-called *frontier risk* thinking,
- too much investment in finding and fixing mistakes can be counterproductive, e.g., customers who do not encounter many fault experiences may be less motivated to renew a maintenance agreement,
- whether the vendor cost of a fault experience includes the user costs associated with the user fault experience, e.g., when software is developed for in-house use. In extreme cases the cost of a fault experience can be hundreds of millions of dollars.[1314]

  With COTS the cost of fault experiences is asymmetric, i.e., the customer bears the cost of the fault experience itself, while the vendor can choose whether to fix the mistake and ship an update. Existing consumer laws provide some legal redress[961] (at least until the existing legal landscape changes[1640]).

  While coding mistakes are exploited by computer viruses, causing business disruption, the greatest percentage of computer related loses come from financial fraud by insiders, and traditional sources of loss such as theft of laptops and mobiles.[1559]

All mistakes have the potential to have costly consequences, but in practice most appear to be an annoyance. One study[37] found that only 2.6% of the vulnerabilities listed in the NVD have been used, or rather their use has been detected, in viruses and network threat attacks on computers.

Figure 6.10 shows the growth in the number of high-risk medical devices, containing the word software in their product summary, achieving premarket approval from the Federal Food and Drug Administration.

The *willingness-to-pay* (WTP) approach to reliability aims to determine the maximum amount that those at risk would individually be willing to pay for improvements to their, or other people's safety. Each individual may only be willing to pay a small amount, but as a group the amounts accumulate to produce an estimated value for the group "worth" of a safety improvement. For instance, assuming that in a group of 1.5 million people, each person is willing to pay £1 for safety improvements that achieve a 1 in 1 million reduction in the probability of death; the summed WTP *Value of Preventing a Statistical Fatality* (VPF) is £1.5 million (the same approach can be used to calculate *Value of Preventing non-fatal Injuries*).

A market has developed for coding mistakes that can be exploited to enable third parties to gain control of other peoples' computers[36] (e.g., spying agencies and spammers), and some vendors have responded by creating vulnerability reward programs. The list of published rewards are both a measure of the value vendors place on the seriousness of particular kinds exploitable mistakes, and the minimum amount the vendor considers sufficient to dissuade discoverers spending time finding someone willing to pay more.

Bountysource is a website where people can pledge a monetary bounty, payable when a specified task is performed. A study by Zhou, Wang, Bezemer, Zou and Hassan[2001] investigated bounties offered to fix specified open issues, of some Github project (the 2,816 tasks had a total pledge value of $365,059). Figure 6.11 shows the total dollar amount pledged for each task; data broken down by issue reporting status of person specifying the task.

The viability of making a living from bug bounty programs is discussed in connection with figure 4.43.

If a customer reports a fault experience, in software they have purchased, what incentive does the vendor have to correct the problem and provide an update (they have the customers' money; assuming the software is not so fault ridden that it is returned for a refund)? Possible reasons include:

- a customer support agreement requires certain kinds of reported faults to be fixed,
- public perception and wanting to maintain customer good will, in the hope of making further sales. One study[75] found that the time taken to fix publicly disclosed vulnerabilities was shorter than for vulnerabilities privately disclosed to the vendor; see section 11.11.3.1,

- a fill-in activity for developers, when no other work is available,
- it would be more expensive not to fix the coding mistake, e.g., the change in behavior produced by the fault experience could cause an accident, or negative publicity, that has an economic impact greater than the cost of fixing the mistake. Not wanting to lose money because a mistake has consequences that could result in a costly legal action (the publicity around a decision driven by a vendors' cost/benefit analysis may be seen as callous, e.g., the Ford Motor Company had to defend itself in court[1098] over its decision not to fix a known passenger safety issue, because they calculated the cost of loss of human life and injury did not exceed the benefit of fixing the issue).

Which implementation mistakes are corrected? While there is no benefit in correcting mistakes that customers are unlikely to experience, it may not be possible to reliably predict whether a mistake will produce a customer fault experience (leading to every mistake having to treated as causing a fault experience). Once a product has been released and known to be acceptable to many customers, there may not be any incentive to actively search for potential fault experiences, i.e., the only mistakes corrected may be those associated with a customer fault reports.

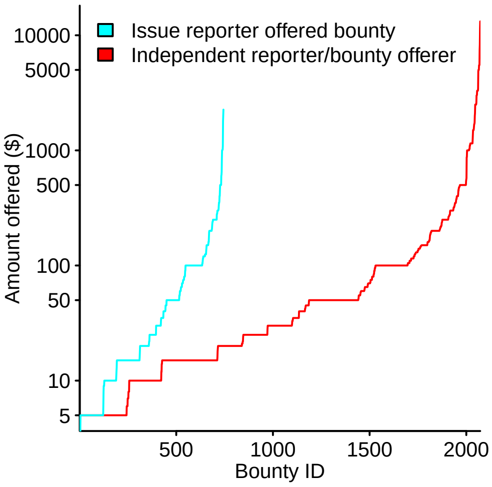


Figure 6.11: Value of bounties offered for 2,816 tasks addressing specified open issues of a Github project; pledges stratified by status of person reporting the pledge issue. Data from Zhou et al.[2001] Github–Local

In some cases applications are dependent on the libraries supplied by the vendor of the host platform. One study[1137] of Apps running under Android found that those Apps using libraries that contained more reported faults had a slightly smaller average user rating in the Google Play Store.

What motivates developers to fix faults reported in Open source projects? Possible reasons include:

- they work for a company that provides software support services for a fee. Having a reputation as the go-to company for a certain bundle of packages is a marketing technique for attracting the attention of organizations looking to outsource support services, or pay for custom modifications to a package, or training.

  Correcting reported faults is a costly signal that provides evidence a company employs people who know what they are doing, i.e., status advertising,
- developers dislike the thought of being wrong or making a mistake; a reported fault may be fixed to make them feel better (or to stop it preying on their mind), also not responding to known problems in code is not considered socially acceptable behavior in some software development circles. Feelings about what constitutes appropriate behavior may cause developers to want to redirect their time to fixing mistakes in code they have written or feel responsible for, provided they have the time; problems may be fixed by developers when management thinks they are working on something else.

6.3 Experiencing a fault

Unintended software behavior is the result of an interaction between a mistake in the code (or more rarely an incorrect translation by a compiler[1124]), and particular input values.

A program's source code may be riddled with mistakes, but if typical user input does not cause the statements containing these mistakes to be executed, the program may gain a reputation for reliability. Similarly, there may only be a few mistakes in the source code, but if they are frequently experienced the program may gain a reputation for being fault-ridden.

Almost all existing research on software reliability has focused on the existence of the mistakes in source code. This is convenience sampling, large amounts of Open source is readily available, while information on the characteristics of program input is very difficult to obtain.

The greater the number of people using a software system, the greater the volume and variety of inputs it is likely to process: consequently there are likely to be more reported fault experiences.

A study by Shatnawi[1660] investigated the impact of the number of sites using a release of telecommunication switch software, on the number of software failures reported. A regression model fitted to the data shows that reported fault experiences decreased over time, and increases with the number of installed sites; see Github–reliability/2014-04-13.R.

A study by Lucente[1153] investigated help desk incident reports, from 800 applications used by a 100,000 employee company with over 120,000 desktop machines. Figure 6.12

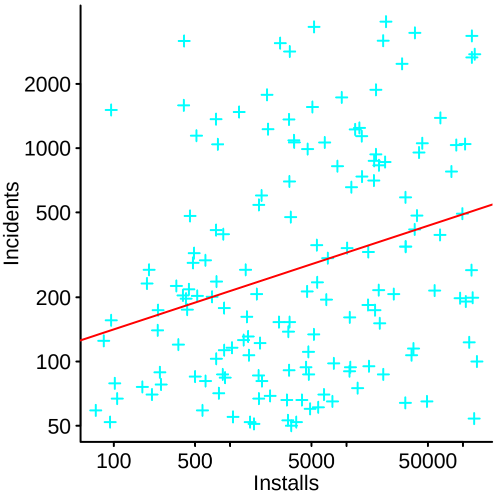


Figure 6.12: Number of incidents reported for each of 800 applications installed on over 120,000 desktop machines; line is fitted regression model. Data from Lucente.[1153] Github–Local

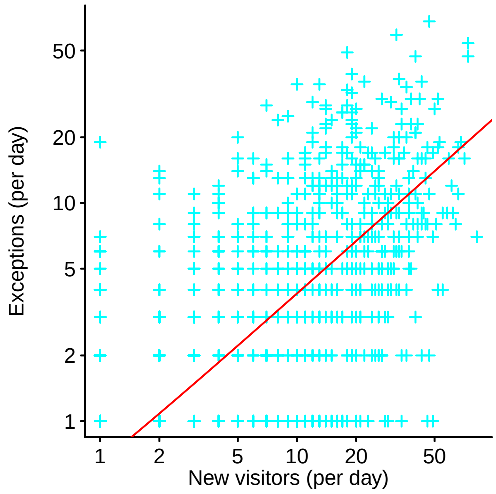


Figure 6.13: Number of exceptions experienced per day against number of new users of the application, for one application prior to its general release; line is a fitted regression model of the form: $Exceptions \propto newUserUses^{0.8}$. Data from Dey et al.[482] Github–Local

shows the number of incidents reported increasing with the number of installs (as is apparent from the plot, the number of installs only explains a small percentage of the variance).

A comparison of the number of fault experiences reported in different software systems[1469] might be used to estimate the number of people using the different systems; any estimate of system reliability has to take into account the volume of usage, and the likely distribution of input values.

The same user input can produce different fault experiences in different implementations of the same functionality, e.g., the POSIX library on 15 different operating systems.[481]

A study by Dey and Mockus[482] investigated the impact of number of users, and time spent using a commercial mobile application, on the number of exceptions the App experienced. The App used Google analytics to log events, which provides daily totals. On many days no exceptions were logged, and when exceptions did occur it is likely that the same set of faults were repeatedly experienced. The fitted regression models (with exceptions as the response variable) contain both user-uses and new-user-uses as power laws, with the exponent for new-user-uses being the largest, and the impact of the version of Android installed on the users' device varied over several orders of magnitude; see Github–reliability/2002-09989.R.

Figure 6.13 shows, for one application on prerelease, the number of exceptions per day for a given number of new users.

6.3.1 Input profile

The environments in which we live, and software systems operate, often experience regular cycles of activity; events are repeated with small variations at a variety of scales (e.g., months of the year, days of the week, and frequent use of the same words;[1094] also see section 2.4.4).

The input profile that results in faults being experienced is an essential aspect of any analysis of program reliability. For instance, when repeatedly adding pairs of floating-point values, with each value drawn from a logarithmic distribution, the likelihood of experiencing an overflow[582] may not be low enough to be ignored (the probability for a single addition overflowing is 5.3×10^{-5} for single precision IEEE and 8.2×10^{-7} for double[v]).

Undetected coding mistakes[vi] exist in shipped systems because the input values needed to cause them to generate a fault experience were not used during the testing process.

Address traces illustrate how the execution characteristics of a program can be dramatically changed by its input. Figure 6.14 shows the number of memory accesses made while executing `gzip` on two different input files. The small colored boxes representing 100,000 executed instruction on the x-axis, and successive 4,096 bytes of stack on the y-axis, the colors denote number of accesses within the given block (using a logarithmic scale).

Mistakes in code that interact with input values that are likely to be selected by developers, during testing, are likely to be fixed during development; beta testing is one method for discovering customer oriented input values that developers have not been testing against. Ideally the input profiles of the test and actual usage are the same, otherwise resources are wasted fixing mistakes that the customer is less likely to experience.

The test process may include automated generation of input values (see section 6.6.2.1).

Does the interaction between mistakes in the source code, and an input profile, generate any recurring patterns in the frequency of fault experiences?

One way of answering this question is to count the number of inputs successfully processed by a program between successive fault experiences.

A study by Nagel and Skrivan[1329] investigated the timing characteristics of fault experiences in three programs, each written independently by two developers. During execution, each program processed inputs selected from the set of permissible values, when a

[v] The general formula is: $\frac{\pi^2}{6\left(\ln \frac{\Omega}{\omega}\right)^2}$ where: Ω and ω are the largest and smallest representable values respectively); the probability of a subtraction underflowing has a more complicated form, but the result differs by at most 1 part in 10^{-9} from the addition formula.

[vi] Management may consider it cost effective to ship a system containing known mistakes.

fault was experienced its identity, execution time up to that point and number of input cases processed were recorded; the coding mistake was corrected and program execution continued until the next fault experience, until five or six faults had been experienced, or the mistake was extremely time-consuming to correct (the maximum number of input cases on any run was 32,808). This cycle was repeated 50 times, always starting with the original, uncorrected, program; the term *repetitive run modeling* was used to denote this form of testing. A later study by Nagel, Scholz and Skrivan[1328] partially replicated and extended this study.

Figure 6.15 shows the order in which distinct faults were experienced by implementation A2, over 50 replications; edge values show the number of times the n^{th} fault experience was followed by a particular fault experience. For example, starting in state `A2-0` fault experience `1` was the first encountered during 37 runs, and this was followed by fault experience 3 during one run.

Figure 6.16, upper plot, shows the number of input cases processed before a given number of fault experiences, during the 50 runs of implementation A2; the lower plot shows the number of inputs processed before each of five distinct fault experiences.

What is the likely number of inputs that have to be processed by implementation A2 for the sixth distinct fault to be experienced? A regression model could be fitted to the data seen in the upper plot of figure 6.16, but a model fitted to this sample of five distinct fault experiences will have a wide confidence interval. There is no reason to expect that the sixth fault will be experienced after processing any number of inputs, there appears to be a change point after the fourth fault, but this may be random noise that has been magnified by the small sample size.

The time and cost of establishing, to a reasonable degree of accuracy, that users of a program have a very low probability of experiencing a fault[280] may not be economically viable.

A study by Dunham and Pierce[509] replicated and extended the work of Nagel and Skriva; problem 1 was independently reimplemented by three developers. The three implementations were each tested with 500,000 input cases, when a fault was experienced the number of inputs processed was recorded, the coding mistake corrected, and program execution restarted. This cycle was repeated four times, always starting with the original implementation, fixing and recording as faults were experienced.

Figure 6.17 shows the number of input cases processed, by two of the implementations (only one fault was ever experienced during the execution of the third implementation), before a given number of fault experiences, during each of the four runs. The grey lines are an exponential regression model fitted to each implementation; these two lines show that as the number of faults experienced grows, more input cases are required to experience another fault, and that code written by different developers has different fault experience rates per input.

A second study by Dunham and Lauterbach[508] used 100 replications for each of the three programs, and found the same pattern of results seen in the first study.

Some published fault experience experiments have used time (computer or user), as a proxy for the quantity of input data. It is not always possible to measure the quantity of input processed, and time may be more readily available.

A study by Wood[1957] analysed fault experiences encountered by a product Q/A group in four releases of a subset of products. Figure 6.18 shows that the fault experience rate is similar for the first three releases (the collection of test effort data for release 4 is known to have been different from the previous releases).

A study by Pradel[1499] searched for thread safety violations in 15 classes in the Java standard library and JFreeChart, that were declared to be thread safe, and 8 classes in Joda-Time not declared to be thread safe; automatically generated test cases were used. Thread safety violations were found in 22 out of the 23 classes; for each case the testing process was run 10 times, and the elapsed time to discover the violation recorded. Figure 6.19 illustrates the variability in the timing of the violations experienced.

A study by Adams[7] investigated reported faults in applications running on IBM mainframes between 1975 and 1980. Figure 6.20 shows that approximately one third of fault experiences first occurred on average every 5,000 months of execution time (over all uses of the product). Only around 2% of fault experiences first occurred after five months of execution time.

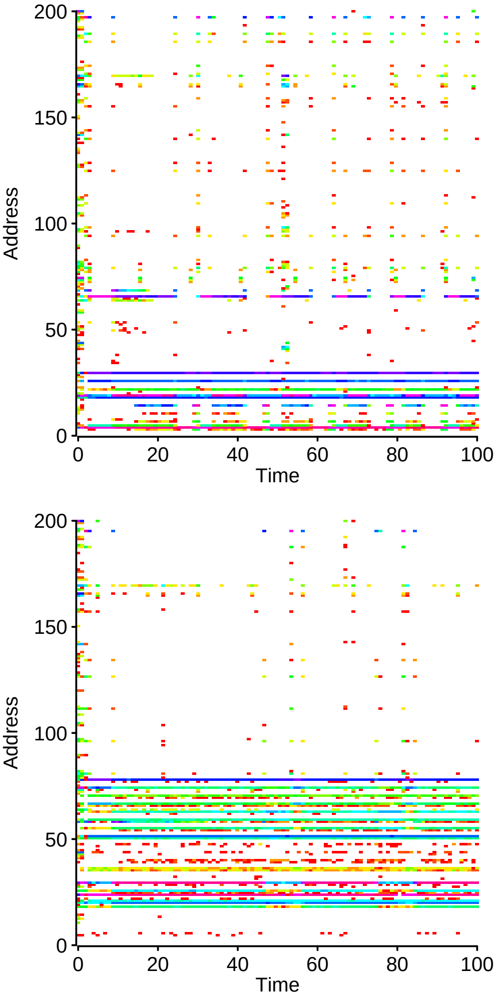


Figure 6.14: Number of accesses to memory address blocks, per 100,000 instructions, when executing `gzip` on two different input files. Data from Brigham Young[250] via Feitelson. Github–Local

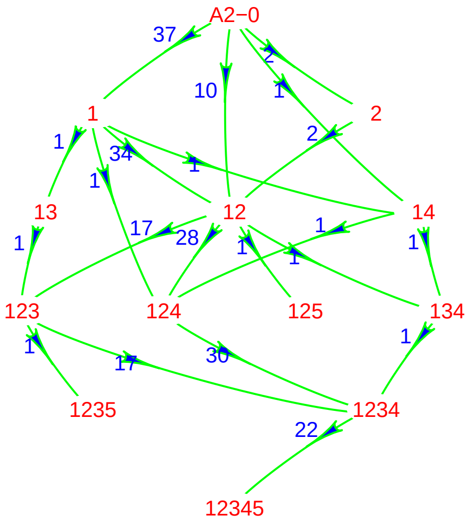
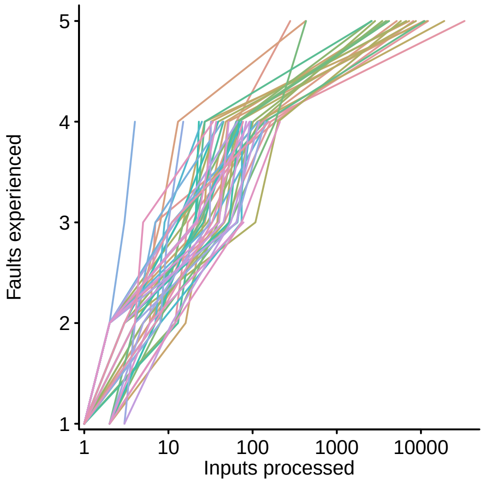


Figure 6.15: Transition counts of five distinct fault experiences in 50 runs of program A2; nodes labeled with each fault experienced up to that point. Data from Nagel et al.[1329] Github–Local



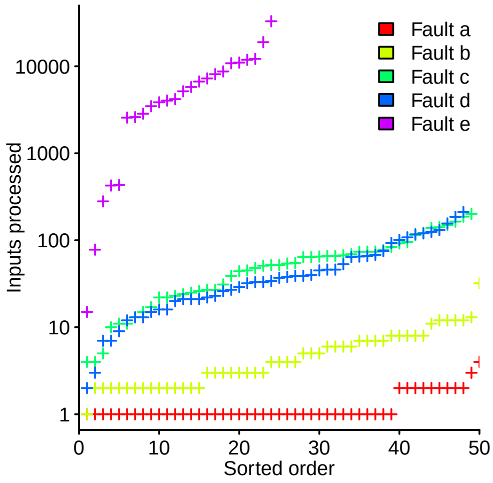


Figure 6.16: Number of input cases processed before a particular fault was experienced by program A2; the list is sorted for each distinct fault. Data from Nagel et al.[1329] Github–Local

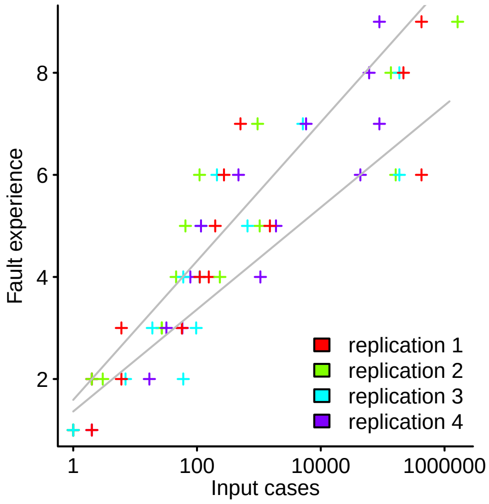


Figure 6.17: Number of input cases processed by two implementations before a fault was experienced, with four replications (each a different color); grey lines are a regression fit for one implementation. Data from Dunham et al.[509] Github–Local

Multiple fault experiences produced by the same coding mistake provide information about the likelihood of encountering input that can trigger that fault experience. Regression models fitted using a biexponential equation (i.e., $a \times e^{b \times x} + c \times e^{d \times x}$, where x is the rank order of occurrences of each fault experience) have been fitted to a variety of program crash data (see fig 11.53).

A study by Zhao and Liu[1995] investigated the crash faults found by fuzzing the files processed by six Open source programs. Figure 6.21 shows the number of unique crash faults experienced by `convert` and `autotrace` (estimated by tracing back to a program location), along with lines fitted using biexponential regression models.

Why is a biexponential model such a good fit? A speculative idea is that the two exponentials are driven by the two independent processes that need to interact to produce a fault experience: the distribution of input values, and the mistakes contained in the source code.[vii]

6.3.2 Propagation of mistakes

The location in the code that triggers a fault experience may appear many executable instructions after the code containing the mistake (that is eventually modified to prevent further the fault experiences).

In some input values a coding mistake may not propagate from the mistake location, to a code location where it can trigger a fault experience. For instance, if variable `x` is mistakenly assigned the value 3, rather than 2, the mistake will not propagate past the condition: `if (x < 8)` (because the behavior is the same for both the correct and mistake value); for this case, an opportunity to propagate only occurs when the mistaken value of `x` changes the value of the conditional test.

How robust is code to small changes to the correct value of a variable?

A study by Danglot, Preux, Baudry and Monperrus[428] investigated the propagation of one-off perturbations in 10 short Java programs (42 to 568 LOC). The perturbations were created by modifying the value of an expression once during the execution of a program (e.g., by adding, or subtracting, one). The effect of a perturbation on program behavior could be to cause it to raise an exception, output an incorrect result, or have no observed effect (i.e., the output is unchanged). Each of a program's selected perturbation points were executed multiple times (e.g., the 41 perturbation points selected for the program `quicksort` were executed between 840 and 9,495 times, per input), with one modification per program execution (requiring `quicksort` to be executed 151,444 times, so that each possible perturbation could occur, for the set of 20 inputs used).

Figure 6.22 shows violin plots for the likelihood that an add-one perturbation has no impact on the output of a program; not all expressions contained in the programs were perturbed, so a violin plot is a visual artefact.

Studies[348] of the impact of soft errors (i.e., radiation induced bit-flips) have found that over 80% of bit-flips have no detectable impact on program behavior.

6.3.3 Remaining faults: closed populations

In a closed population no coding mistakes are added (e.g., no new code is added) or removed (i.e., reported faults are not fixed), and the characteristics of the input distribution remain unchanged.

After N distinct faults have been experienced, what is the probability that there exists new, previously unexperienced, faults?

Data on reported faults commonly takes two forms: incidence data (i.e., a record of the date of first report, with no information on subsequent reports involving the same fault experience), and abundance data (i.e., a record of every fault experience).

Software reliability growth has often been modeled as a nonhomogeneous Poisson process, with researchers fitting various formulae to small amounts of incidence data.[1115] Unfortunately, it is not possible to use one sample of incidence data to distinguish between different exponential order growth models[1266] (i.e., this data does not contain enough

[vii] Working out which process corresponds to which exponential appearing in the plots is left as an exercise to the reader (because your author has no idea).

information to do the job asked of it). It is often possible to fit a variety of equations to fault report data, using regression modeling: however, predictions about future fault experiences made using these models is likely to be very unreliable (see fig 11.50).

When abundance data is available, the modeling approach discussed in section 4.3.2 can be used to estimate the number of unique items within a population, and the number of new unique items likely to be encountered with additional sampling.

A study by Kaminsky, Eddington and Cecchetti[954] investigated crash faults in three releases of Microsoft Office and OpenOffice (plus other common document processors), produced using fuzzing. Figure 6.23 shows actual and predicted growth in crash fault experiences in the 2003, 2007 and 2010 releases of Microsoft Office, along with 95% confidence intervals. Later versions are estimated to contain fewer crash faults, although the confidence interval for the 2010 release is wide enough to encompass the 2007 rate.

Figure 6.24 shows the number of duplicate crashes experienced when the same fuzzed files were processed by the 2003, 2007 and 2010 releases of Microsoft Office. The blue/purple lines are the two components of fitted biexponential models for the three curves.

The previous analysis is based on information about faults that have been experienced. What is the likelihood of a fault experience, given that no faults have been experienced in the immediately previous time, T?

An analysis by Bishop and Bloomfield[199] derived a lower bound for the reliability function, R, for a program executing without experiencing a fault for time t, after it has executed for time T without failure; it is assumed that the input profile does not change during time $T+t$. The reliability function is:

$$R(t|T) \geq 1 - \frac{t}{T+t}e^{-\frac{T}{t}\log(1+\frac{t}{T})}$$

If t is much smaller than T, this equation can be simplified to: $R(t|T) \geq 1 - \dfrac{t}{(T+t)\times e}$

For instance, if a program is required to execute for 10 hours with reliability 0.9999, the initial failure free period, in hours, is:

$$0.9999 \geq 1 - \frac{10}{(T+10)\times e}$$

$$T \geq \frac{10}{(1-0.9999)\times e} - 10 \approx 36,778$$

If T is much smaller than t, the general solution can be simplified to: $R(t|T) \geq \dfrac{T}{t}$

How can this worst case analysis be improved on?

Assuming a system executes N times without a failure, and has a fixed probability of failing, p, the probability of one or more failures occurring in N executions is given by:

$$C = \sum_{n=1}^{N} p(1-p)^{n-1} = p\frac{1-(1-p)^N}{1-(1-p)} = 1-(1-p)^N$$

How many executions, without failure, need to occur to have a given confidence that the actual failure rate is below a specified level? Rearranging, gives: $N = \left\lceil \dfrac{\log(1-C)}{\log(1-p)} \right\rceil$

Plugging in values for confidence, $C = 0.99$, and failure probability, $p < 10^{-4}$, then the system has to execute without failure for 46,050 consecutive runs.

This analysis is not realistic because it assumes that the probability of failure, p, remains constant for all input cases; studies show that p can vary by several orders of magnitude.

6.3.4 Remaining faults: open populations

In an evolving system, existing coding mistakes are corrected and new ones are made; new features may be added that interact with existing functionality (i.e., there may be a change of behavior in the code executed for the same input), and the user base is changing (e.g., new users arrive, existing users leave, and the number of users running a particular version changes as people migrate to a newer release); the population of mistakes is open. Studies of fault reports in an open population that fail to take into account the impact of the time varying population[808] will not produce reliable results.

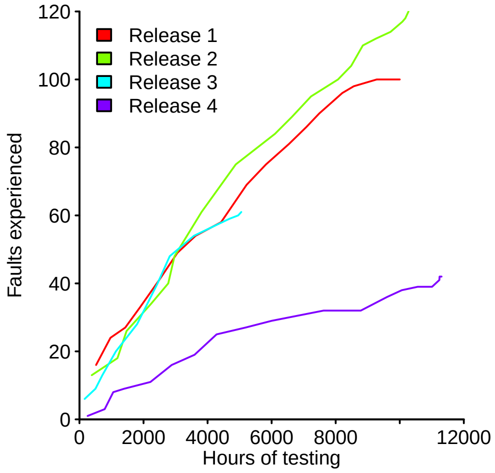


Figure 6.18: Faults experienced against hours of testing, for four releases of a product. Data from Wood.[1957] Github–Local

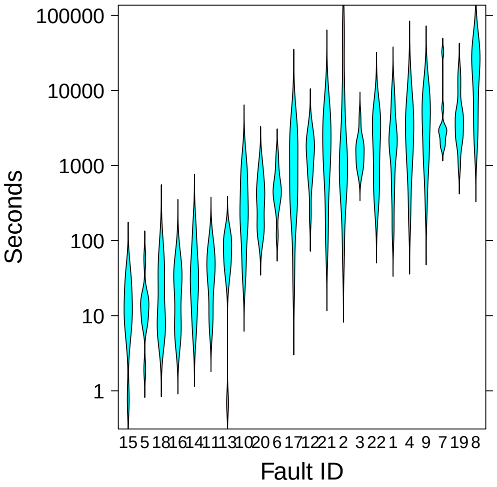


Figure 6.19: Time taken to encounter a thread safety violation in 22 Java classes, violin plots for 10 runs of each class. Data kindly supplied by Pradel.[1499] Github–Local

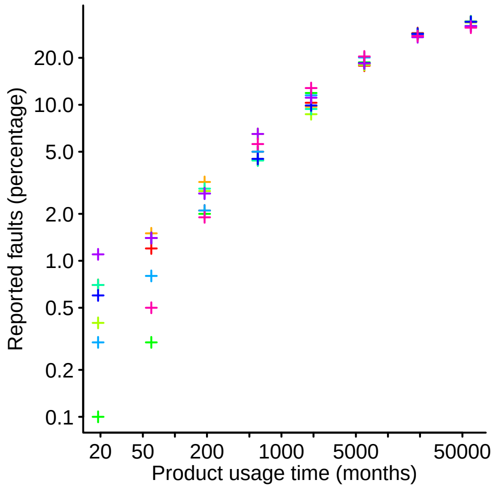


Figure 6.20: Percentage of fault experiences having a given mean time to first experience (in months, over all installations of a product), for nine products. Data from Adams.[7] Github–Local

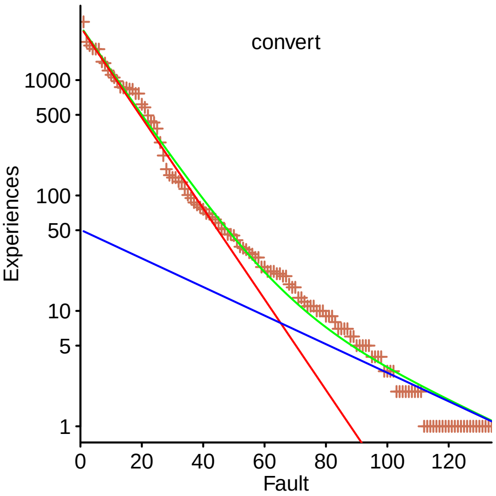


Figure 6.21: Number of times the same fault was experienced in one program, crashes traced to the same program location; with fitted biexponential equation (green line; red/blue lines the two components). Data kindly provided by Zhao.[1995] Github–Local

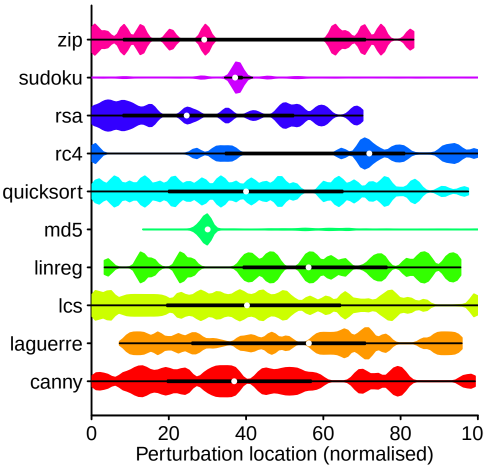


Figure 6.22: Violin plots of likelihood (local y-axis) that an add-one perturbation at a (normalised) program location will not change the output behavior. Data from Danglot et al.[428] Github–Local

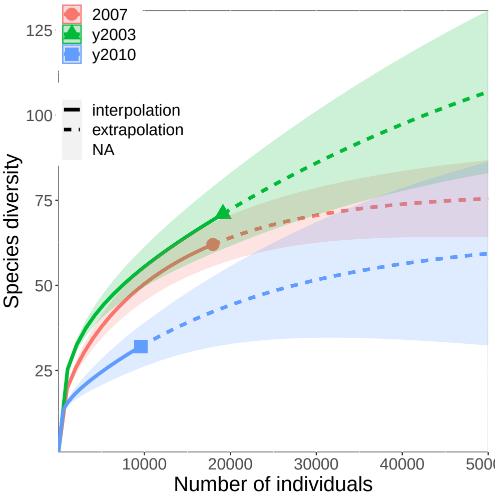


Figure 6.23: Predicted growth, with 95% confidence intervals, in the number of new crash fault experiences in the 2003, 2007 and 2010 releases of Microsoft Office. Data from Kaminsky et al.[954] Github–Local

Section 4.3.2.2 discusses estimation in open populations.

What form of regression models can be fitted to data on fault reports from an open population?

A study by Sun, Le, Zhang and Su[1772] investigated the fault reports for GCC and LLVM. Figure 6.25 shows the number of times a distinct mistake has been responsible for a fault report in GCC (from 1999 to 2015), with a fitted biexponential, and its component exponentials.

A study by Sadat, Bener and Miranskyy[1603] investigated duplicate fault reports in Apache, Eclipse and KDE over 18-years. Figure 6.26 shows the number of times distinct faults reported in KDE, with a fitted triexponential (green) and the three component exponentials.

Being able to fit this form of model suggests a pattern that may occur in other collections of reported faults, but there is no underlying theory.

Successive releases of a software system often include a large percentage of code from earlier releases. The collection of source code that is new in each release can be treated as a distinct population containing a fixed number of mistakes; these populations do not grow but can shrink (when code is deleted). The code contained in the first release is the foundation population.

The number of faults that could be experienced in a version of a software system is the sum of the estimated fault experiences that could be triggered by the mistakes in the code it contains, from the current and earlier releases.

A study by Massacci, Neuhaus and Nguyen[1202] investigated 899 Security Advisories in Firefox, reported against six major releases. Their raw data is only available under an agreement that does not permit your author to directly distribute it to readers; the data used in the following analysis was reverse engineered from the paper, or extracted by your author from other sources.

The following analysis attempts to build a model of the relationship between the age of code, end-user source code usage and reported faults.

Table 6.2 shows the lowest version (columns) of Firefox containing a known mistake in the source code, and the highest version (rows) to which a corresponding fault report exists. For instance, 42 faults were discovered in version 2.0 corresponding to mistakes made in the source code written for version 1.0. Only corrected coding mistakes have been counted, unfixed mistakes are not included in the analysis. Each version of Firefox has a release, and retirement date, after which the version is no longer supported, i.e., no more coding mistakes are corrected in the retired version.

	1.0	**1.5**	**2.0**	**3.0**	**3.5**	**3.6**
1.0	79					
1.5	71	108				
2.0	42	104	126			
3.0	97	15	22	67		
3.5	32			30	32	
3.6	13		1	5	41	14

Table 6.2: Number of reported security advisories in versions of Firefox; coding mistake made in version columns, advisory reported in version row. Data from Massacci et al.[1202]

How many users does each version of Firefox have over time?

Figure 6.28 shows the market share of the six versions of Firefox between official release and end-of-support. Estimated values appear to the left of the vertical grey line, values from measurements to the right; note: at its end-of-support date version 2.0 still had a significant market share.

The analysis assumes that every user of the Internet uses a browser; figure 6.29 shows the growth of internet usage over time, broken down by nation development status.

The end-user usage of source code originally written for a particular version of Firefox, over time, is calculated as follows: (number of lines of code originally written for a particular version contained within the code used to build a later version, or that particular version; call this the build version) multiplied by (the market share of the build version) multiplied by (the number of Internet users, based on the developed world count).

Figure 6.30 is based on treating the source code originally written for Firefox version 1.0 as the foundation code. The yellow points are the code usage for version 1.0 code executing in Firefox build version 1.0, the green points the code usage for version 1.0 code executing in build version 1.5 and so on. The red points show the sum of version 1.0 code usage over all build versions.

Much of the overall growth comes from growth in Internet usage, and in the early years there is also substantial growth in browser market share.

This analysis assumes that the browsing habits of people who started using the Internet in 2004 are the same as those who first started in 2010, and the propensity to report a fault experience is unchanged (it also ignores cultural differences, e.g., European users vs. Chinese users). Changes in the content of web pages could also have some effect on which components of Firefox are executed.

6.4 Where is the mistake?

Information about where mistakes are likely to be made can be used to focus problem solving resources in those areas likely to produce the greatest returns. A few studies[417] have measured across top-level entities such as project phase (e.g., requirements, coding, testing, documentation), while others have investigated specific components (e.g., source code, configuration file), or low level constructs (e.g., floating-point[483]).

The root cause of a mistake, made by a person, may be knowledge based (e.g., lack of knowledge about the semantics of the programming language used), rule based (e.g., failure to correctly apply known coding rules), or skill based (e.g., fail to copy the correct value of a numeric constant in an assignment).[1545]

Mistakes in hardware[334, 498, 880] tend to occur much less frequently than mistakes in software, and mistakes in hardware are not considered here[viii]

The *user interface*, the interaction between people and an application, can be a source of fault experiences in the sense that a user misinterprets correct output, or selects a parameter option that produces unintended program behavior. User interface issues are not considered here.

Accidents where a large loss occurs (e.g., fatalities) are often followed by an accident investigation. The final report produced by the investigation may involve language biases that affect what has been written, and how it may be interpreted.[1873]

A study by Brown and Altadmri[262] investigated coding mistakes made by students, and the beliefs their professors had about common student mistakes; the data came from 100 million compilations across 10 million programming sessions using Blackbox (a Java programming environment). There was very poor agreement between professor beliefs and the actual ranked frequency of student mistakes; see Github–reliability/educators.R.

Software supply chains are a target for criminals seeking to infect computers with malicious software;[1390] an infected software update may be downloaded and executed by millions of users

6.4.1 Requirements

The same situation can be approached from multiple viewpoints, depending on the role of the viewer; see fig 2.49. Those implementing a system may fail to fully appreciate all the requirements implied by the specification; context is important, see fig 2.48.

A requirements mistake is made when one or more requirements are incorrect, inconsistent or incomplete; an ambiguous specification[186] contains potential mistakes. The number of mistakes contained in requirements may be of the same order of magnitude,[417] or exceed, the number of mistakes found in the code;[1530] different people bring different perspectives to requirements analysis, which can result in them discovering different problems.[1085]

Software systems are implemented by generating and interpreting language (human and programming). Reliability is affected by human variability in the use of language,[189] what

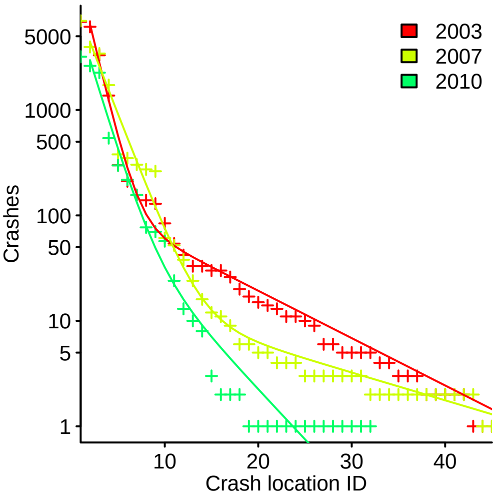


Figure 6.24: Number of crashes traced to the same executable location (sorted by number of crashes), in the 2003, 2007 and 2010 releases of Microsoft Office; lines are fitted biexponential regression models. Data from Kaminsky et al.[954] Github–Local

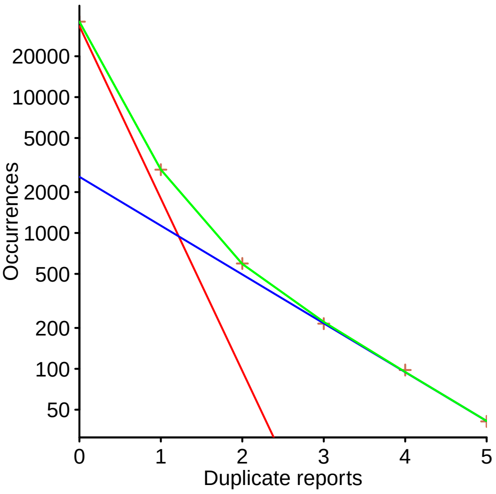


Figure 6.25: Number of occurrences of the same mistake responsible for a reported fault in GCC, with fitted biexponential regression model, and component exponentials. Data from Sun et al.[1772] Github–Local

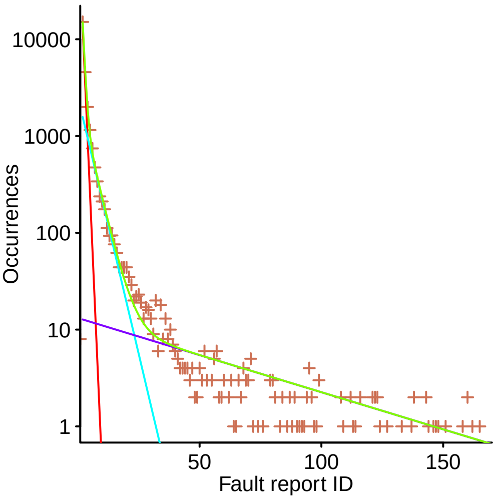


Figure 6.26: Number of instances of the same reported fault in KDE, with fitted triexponential regression model. Data from Sadat et al.[1603] Github–Local

[viii] Your author once worked on a compiler for a cpu that was still on alpha release; the generated code was processed by a sed script to handle known problems in the implementation of the instruction set, problems which changed over time as updated versions of the cpu chip became available.

individuals consider to be correct English syntax,[1730] and the interpretation of numeric phrases. Language issues are discussed in section 6.1.4 and section 6.4.2.

During the lifetime of a project, existing requirements are misinterpreted or changed, and new requirements are added.

Non-requirement mistakes may be corrected by modifying the requirements; see fig 8.28. In cases where a variety of behaviors are considered acceptable, modifying the requirements documents may be the most cost effective path to resolving a mistake.

A study by van der Meulen, Bishop and Revilla[1858] investigated the coding mistakes made in 29,000 implementations of the $3n+1$ problem (the programs had been submitted to a programming contest). All submitted implementations were tested, and programs producing identical outputs were assigned to the same equivalence class (competitors could make multiple submissions, if the first failed to pass all the tests). In many cases the incorrect output, for an equivalence class, could be explained by a failure of the competitor to implement a requirement implied by the problem being solved, e.g., failing to swap input number pairs, when the first was larger than the second.

Figure 6.31 shows the 36 equivalence classes containing the most members; the most common is the correct output, followed by always returning 0 (zero).

Studies[350] have found that people take longer to answer question involving a negation, and are less likely to give a correct answer.

A study by Winter, Femmer and Vogelsang[1951] investigated subject performance on requirements expressed using affirmative and negative quantifiers. Subjects saw affirmative (e.g., “All registered machines must be provided in the database.”) and negative (e.g., “No deficit of a machine is not provided in the database.”) requirements, and had to decide which of three situations matched the sentence. Affirmative wording had a greater percentage of correct answers in four of the nine quantifier combinations (negative wording had a higher percentage of correct answers for the quantifiers: `All but` and `More than`).

Figure 6.32 shows the response time for each quantifier, broken down by affirmative/negative. Average response time for negative requirements was faster for two, of the nine, quantifiers: `All but` and `None` (there was no statistical difference for `At most`).

The minimum requirements for some software (e.g., C and C++ compilers) is specified in an ISO Standard. The ISO process requires that the committee responsible for a standard maintain a log of potential defect submissions received, along with the committee’s response. Figure 6.33 shows the growth of various kinds of defects reported against the POSIX standard.[888]

There have been very few studies[1671] of the impact of the form of specification on its implementation.

Two studies[355,915] have investigated the requirements coverage achieved by two compiler validation suites.

Source code is written to implement a requirement, or to provide support for code that implements requirements. An `if-statement` represents a decision and each of these decisions should be traceable to a requirement or an internal housekeeping requirement. A project by Jones and Corfield[916] cross-referenced the `if-statements` in the source of a C compiler to every line in the 1990 version of the C Standard.[1615] Of the 53 files containing references to either the C Standard or internal documentation, 13 did not contain any references to the C Standard (for the 53 files the average number of references to the C Standard was 46.6). The average number of references per page of the language chapter of the Standard was approximately 14. For more details see Github–projects/Model-C/.

6.4.2 Source code

Source code is the focus of much software engineering research: it is the original form of an executable program that produces fault experiences, and is usually what is modified by developers to correct mistakes. From the research perspective it is now available in bulk, and techniques for analysing it are known, and practical to implement.

There are recurring patterns in the changes made to source code to correct mistakes,[1423] one reason for this is that some language constructs are used much more often than others.[919] The idea that there is an association between fault reports and particular usage

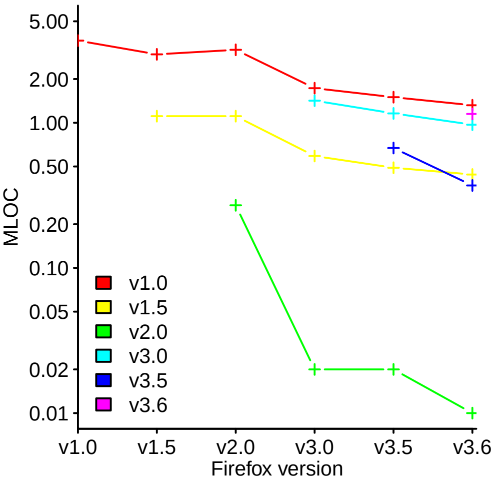


Figure 6.27: Lines of source in early versions of Firefox, broken down by the version in which it first appears. Data extracted from Massacci et al.[1202] Github–Local

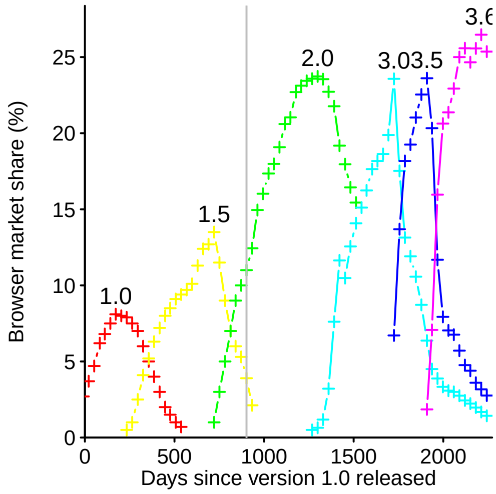


Figure 6.28: Market share of Firefox versions between official release and end-of-support (left of grey line are estimates, right are measurements). Data from Jones.[477] Github–Local

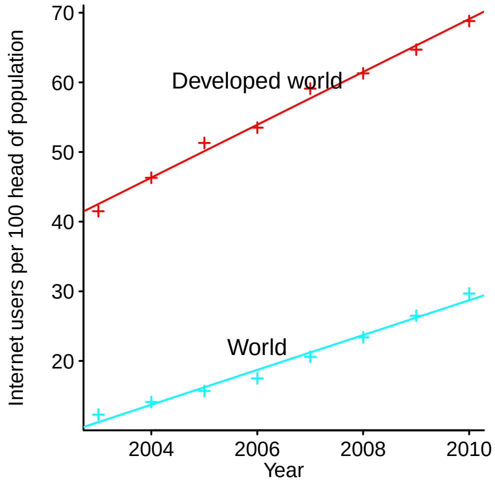


Figure 6.29: Number of people with Internet access per 100 head of population in the developed world, and the whole world; lines are fitted regression models. Data from ITU.[881] Github–Local

patterns in source code, or program behavior, is popular; over 40 association measures have been proposed.[1155]

Errors of omission can cause faults to be experienced. One study[748] of error handling by Linux file systems found that possible returned error codes were not checked for 13% of function calls, i.e., the occurrence of an error was not handled. Cut-and-paste is a code editing technique that is susceptible to errors of omission, that is, failing to make all the necessary modifications to the pasted version;[165] significant numbers of cut-and-paste errors have been found in JavaDoc documentation.[1409]

Common patterns of mistakes are also seen in the use of programming language syntax and semantics. Figure 6.34 shows ranked occurrences of each kind of compiler message generated by Java and Python programs, submitted by students.

Proponents of particular languages sometimes claim that programs written in the language are more reliable (other desirable characteristics may also be claimed), than if written in other languages. Most experimental studies comparing the reliability of programs written in different languages have either used students,[683] or had low statistical power. A language may lack a feature that, if available and used, would help to reduce the number of mistakes made by developers, e.g., support for function prototypes in C,[1651] which were added to the language in the first ANSI Standard.

To what extent might programs written in some languages be more likely to appear to behave as expected, despite containing mistakes?

A study by Spinellis, Karakoidas and Lourida[1727] made various kinds of small random changes to 14 different small programs, each implemented in 10 different languages (400 random changes per program/language pair). The ability of these modified programs to compile, execute and produce the same output as the unmodified program was recorded.

Figure 6.35 shows the fraction of programs that compiled, executed and produced correct output, for the various languages. There appear to be two distinct language groupings, each having similar successful compilation rates; one commonality of languages in each group is requiring, or not, variables to be declared before use. One fitted regression model (see Github–reliability/fuzzer/fuzzer-mod.R) contains an interaction between language and program (the problems implemented did not require many lines of code, and in some cases could be solved in a single line in some languages), and a logarithmic dependency on program length (i.e., number of lines).

Other studies[142] have investigated transformations that modify a program without modifying its behavior.

A study by Aman, Amasaki, Yokogawa and Kawahara[47] investigated time-to-bug-fix events, in the source files of 50 projects implemented in Java and 50 in C++ . The survival time of files (i.e., time to fault report causing the source to be modified) was the same for both languages, and number of developers, and number of project files; they had almost zero impact on a Cox model fitted to the data; see Github–survival/Profes2017-aman.R.

Various metrics have been proposed as measures of some desirable, or undesirable, characteristic of a unit of code, e.g., a function. Halstead's and McCabe's cyclomatic complexity are perhaps the most well-known such metrics (see section 7.2.11), both count the source contained within a single function. Irrespective of whether these metrics strongly correlate with anything other than lines of code,[1065] they can be easily manipulated by splitting functions with high values into two or more functions, each having lower metric values (just as it is possible to reduce the number of lines of code in a function, by putting all the code on one line).

The value of McCabe's complexity (number of decisions, plus one) for the following function is 5, and there are 16 possible paths through the function:

```
int main(void)
{
if (W) a(); else b();
if (X) c(); else d();
if (Y) e(); else f();
if (Z) g(); else h();
}
```

each `if...else` contains two paths and there are four in series, giving $2 \times 2 \times 2 \times 2$ paths. Restructuring the code, as below, removes the multiplication of paths caused by the sequences of `if...else`:

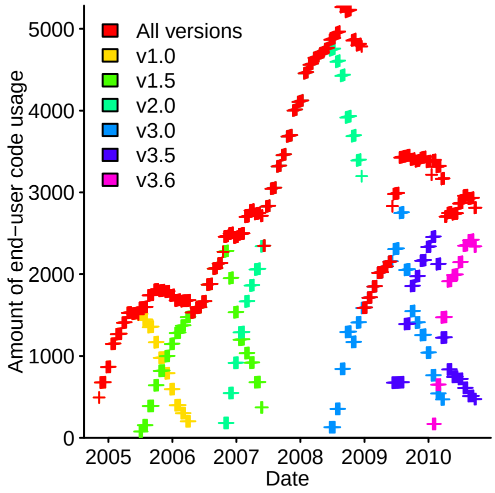


Figure 6.30: Amount of end-user usage of code originally written for Firefox version 1.0, by various other versions; red is sum over all versions. Data extracted from Massacci et al.[1202] Github–Local

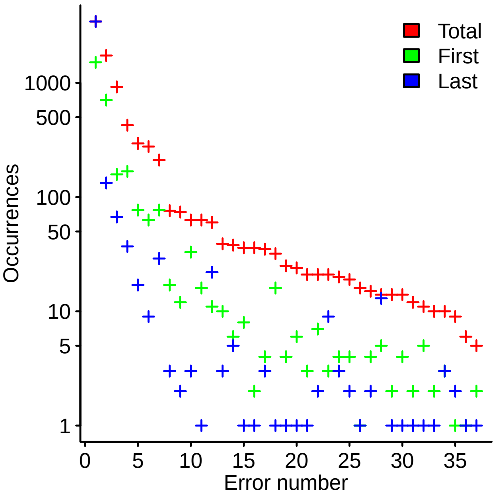


Figure 6.31: Total number of implementations in each of 36 equivalence classes, plus both first and last competitor submissions. Data from van der Meulen et al.[1858] Github–Local

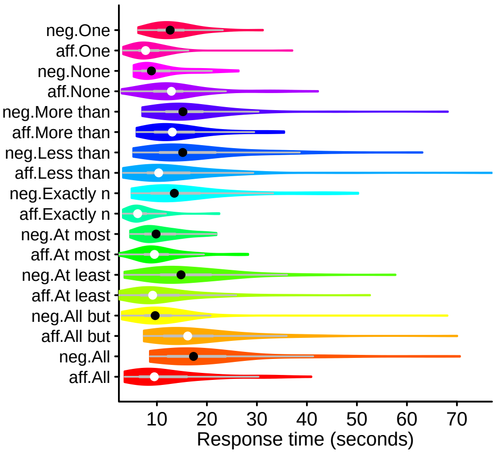


Figure 6.32: Violin plot of the time taken to response to a question about a requirement, for nine quantifiers paired by affirmative/negative. Data from Winter et al.[1951] Github–Local

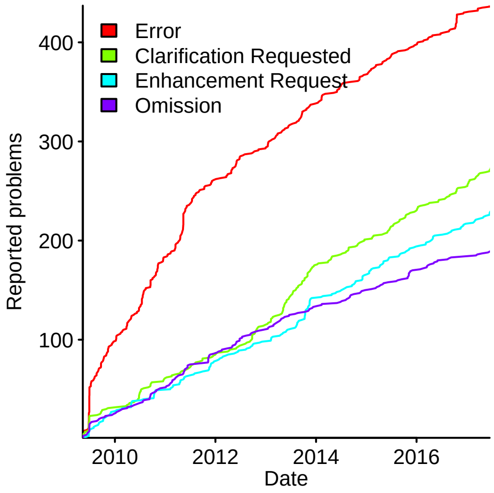


Figure 6.33: Cumulative number of potential defects logged against the POSIX standard, by defect classification. Data kindly provided by Josey.[1398] Github–Local

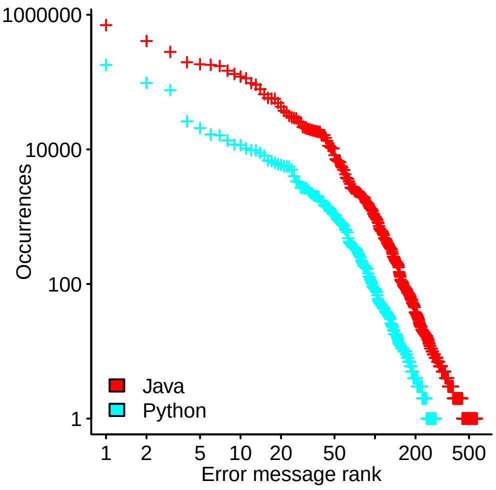


Figure 6.34: Ranked occurrences of compiler messages generated by student submitted Java and Python programs. Data from Pritchard.[1511] Github–Local

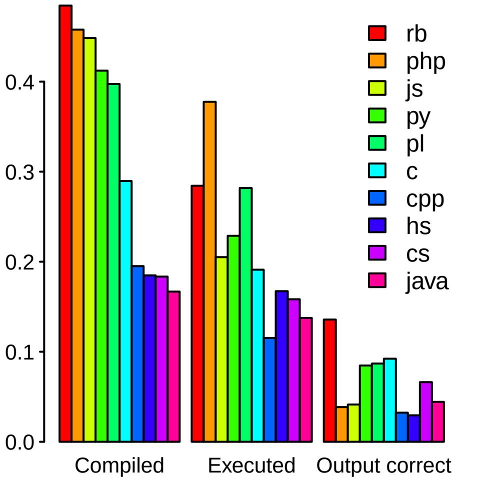


Figure 6.35: Fraction of mutated programs, in various languages, that successfully compiled/executed/produced the same output. Data from Spinellis et al.[1727] Github–Local

```
void a_b(void)
              {if (W) a(); else b();}
void c_d(void)
              {if (X) c(); else d();}
void e_f(void)
              {if (Y) e(); else f();}
void g_h(void)
              {if (Z) g(); else h();}

int main(void)
{
a_b();
c_d();
e_f();
g_h();
}
```

reducing the McCabe complexity of `main` to 1, with the four new functions each having a McCabe complexity of two. Where has the complexity that `main` once had, gone? It now *exists* in the relationship between the functions, a relationship that is not included in the McCabe complexity calculation; the number of paths that can be traversed, by a call to `main` at runtime, has not changed, but a function based count now reports one path.

A metric that assigns a value to individual functions (i.e., its value is calculated from the contents of single functions) cannot be used as a control mechanism (i.e., require that values not exceed some limit), because its value can be easily manipulated by moving contents into newly created functions. The software equivalent of what is known as *accounting fraud* in accounting.

Predictions are sometimes attempted[1675, 2008] at the file level of granularity, e.g., predicting which files are more likely to be the root cause of fault experiences; the idea being that the contents of highly ranked files be rewritten. Any reimplementation will include mistakes, and the cost of rewriting the code may be larger than handling the fault reports in the original code, as they are discovered.

The idea that there is an optimal value for the number of lines of code in a function body has been an enduring meme (when object-oriented programming became popular, the meme mutated to cover optimal class size). See fig 8.39 for a discussion of the U-shaped defect density paper chase; other studies[1025] investigating the relationship between reported fault experiences and number of lines of code, have failed to include program usage information (i.e., number of people using the software) in the model.

The fixes for most user fault reports involve changing a few lines in a single function, and these changes occur within a single file. A study[705] of over 1,000 projects for each of C, Java, Python and Haskell found that correcting most coding mistakes involved adding and deleting a few tokens.

A study by Lucia[1154] investigated fault localization techniques. Figure 6.36 shows the percentage of fault reports whose correction involved a given number of files, modules or lines; lines are power laws fitted using regression.

A study by Zhong and Su[1999] investigated commits associated with fault reports in five large Open source projects. Figure 6.37 shows the number of files modified while fixing reported faults, against normalized (i.e., each maximum is 100) number of commits made while making these changes.

One study[1358] found that 36% of mistakes logged during development were made in phases that came before coding (Team Software Process was used and many of the mistakes may have been minor); see Github–reliability/2018_005_defects.R.

When existing code is changed, there is a non-zero probability of a mistake being made.

A study by Purushothaman and Perry[1515] investigated small source code changes made to one subsystem of the 5ESS telephone switch software (4,550 files containing almost 2MLOC, with 31,884 modification requests after the first release changing 4,293 of these files). Figure 6.38 is based on an analysis of fault reports traced to updates, that involved modifying/inserting a given number of lines, and shows the percentage of each kind of modification that eventually led to a fault report.

Using configuration files to hold literal values that would otherwise need to be present in the source code can greatly increase runtime flexibility. Mistakes in the use of configuration options can lead to fault experiences; see Github–reliability/fse15.R.

6.4.3 Libraries and tools

Libraries provide functionality believed to be of use to many programs, or that is difficult to correctly implement without specialist knowledge (e.g., mathematical functions). Many language specifications include a list of functions that conforming implementations are required to support.

The implementation of some libraries requires domain specific expertise, e.g., the handling of branch cuts in some maths functions.[1679] Some library implementations may contain subtle, hard to detect mistakes: for instance, random number generators may generate sequences containing patterns[1089] that create spurious correlations in the results; even widely used applications can suffer from this problem, e.g., Microsoft Excel.[1220]

The availability of many open source libraries can make it more cost effective to use third-party code, rather than implementing a bespoke solution.

A study by Decan, Mens and Constantinou[462] investigated the time taken for security vulnerabilities in packages hosted in the npm repository to be fixed, along with the time taken to update packages that depended on a version of a package having a reported vulnerability. Figure 6.39 shows survival curves (with 95% confidence bounds), for high and medium severity vulnerabilities, of time to fix a reported vulnerability (Base), and time to update a dependency (Depend) to a corrected version of a package.

The mistake that leads to a fault experience may be in the environment in which a program is built and executed. Many library package managers support the installation of new packages via a command line tool. One study[1828] made use of typos in the information given to command line package managers to cause a package other than the one intended to be installed.

Compilers, interpreters and linkers are programs, and contain mistakes (e.g., see fig 6.25), and the language specification may also be inconsistent or under specified, e.g., ML.[949]

Different support tools may produce different results, e.g., statement coverage,[1970] and call graph construction (see fig 13.1).

6.4.4 Documentation

Documentation is a cost paid today, that is intended to provide a benefit later for somebody else, or the author.

If user documentation specifies functionality that is not supported by the software, the vendor may be liable to pay damages to customers expecting to be able to perform the documented functionality.[960] Non-existent documentation is not unreliable, but documentation that has not been updated to match changes to the software is.

There have been relatively few studies of the reliability of documentation. A study by Rubio-Gonzalez and Libit[1596] investigated the source code of 52 Linux file systems, which invoked 42 different system calls and returned 30 different system error codes. The 871 KLOC contained 1,784 instances of undocumented error return codes; see Github–projects/err-code_mismatch.R. A study by Ma, Liu and Forin[1166] tested an Intel x86 cpu emulator and found a wide variety of errors in the documentation specifying the behavior of the processor.

The Microsoft Server protocol documents[1260] sometimes specify that the possible error values returned by an API function are listed in the Windows error codes document, which lists over 2,500 error codes.

6.5 Non-software causes of unreliability

Hardware contains moving parts that wear out. Electronic components operate by influencing the direction of movement of electrons; when the movement is primarily in one direction atoms migrate in that direction, and over time this migration degrades device operating characteristics.[1733] Fabricating devices with smaller transistors decreases mean time to failure; expected chip lifetimes have dropped from 10 years to 7, and continue to decrease.[1927]

As the size of components shrinks, and the number of components on a device increases, the probability that thermal noise will cause a bit to change state increases.[999]

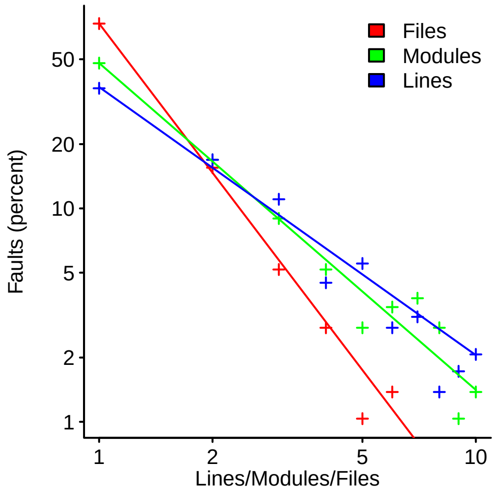


Figure 6.36: Number of fault reports whose fixes involved a given number of files, modules or lines in a sample of 290 faults in AspectJ; lines are fitted power laws. Data from Lucia.[1154] Github–Local

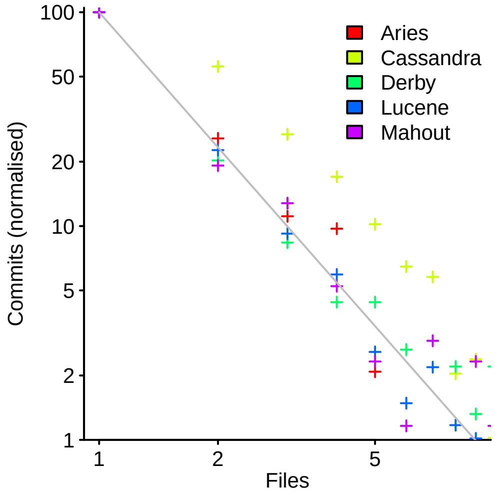


Figure 6.37: Normalized number of commits (i.e., each maximum is 100), made to address fault reports, involving a given number of files in five software systems; grey line is representative of regression models fitted to each project, and has the form: $Commits \propto Files^{-2.1}$. Data from Zhong et al[1999] via M. Monperrus. Github–Local

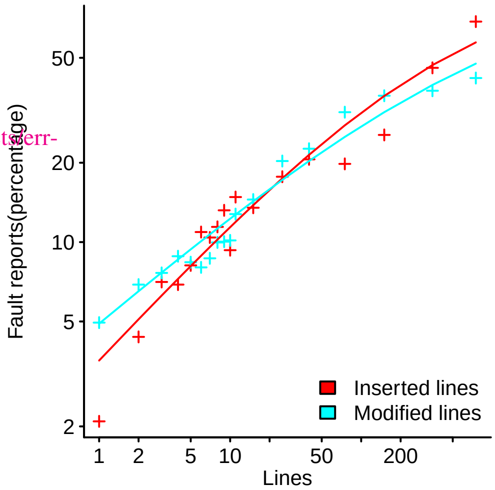


Figure 6.38: Percentage of insertions/modifications of a given number of lines resulting in a reported fault; lines are fitted beta regression models of the form: $percent\_faultReports \propto \log(Lines)$. Data from Purushothaman et al.[1515] Github–Local

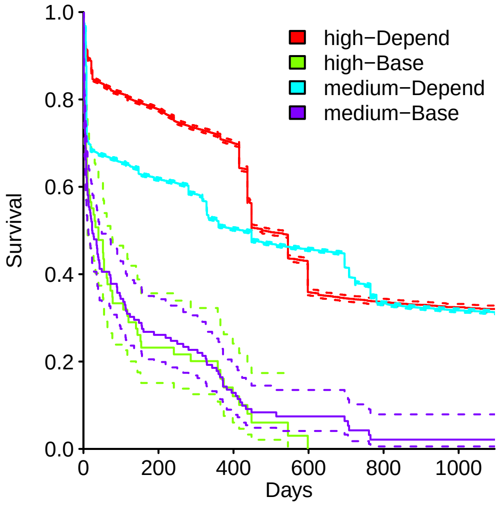


Figure 6.39: Survival curve (with 95% confidence bounds) of time to fix vulnerabilities reported in npm packages (Base) and time to update a package dependency (Depend) to a corrected version (i.e., not containing the reported vulnerability); for vulnerabilities with severity high and medium. Data from Decan et al.[462] Github–Local

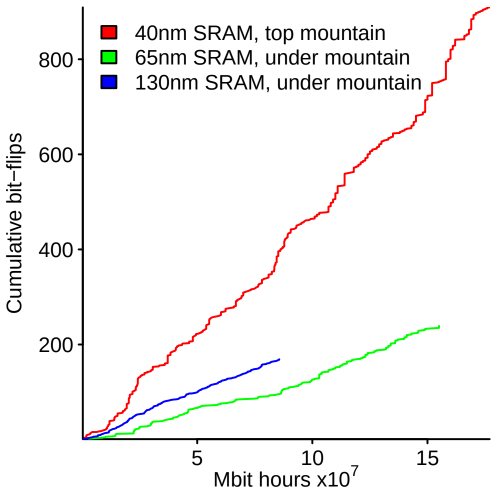


Figure 6.40: Number of bit-flips in SRAM fabricated using various processes, with devices on top of, or under a mountain in the French Alps. Data kindly provided by Autran.[91] Github–Local

Faulty hardware does not always noticeably change the behavior of an executing program; apparently correct program execution can occur in the presence of incorrect hardware operation, e.g., image processing.[1373] Section 6.3.2 discusses studies showing that many mistakes have no observable impact on program behavior.

For a discussion of system failure traced to either cpu or DRAM failures see table 10.7, and for a study investigating the correlation between hardware performance and likelihood of experiencing intermittent faults see section 10.2.3.

A software reliability problem rarely encountered outside of science fiction, a few decades ago, now regularly occurs in modern computers: cosmic rays (plus more local sources of radiation, such as the materials used to fabricate devices) flipping the value of one or more bits in memory, or a running processor. Techniques for mitigating the effects of radiation induced events have been proposed.[1321]

The two main sources of radiation are alpha-particles generated within the material used to fabricate and package devices, and Neutrons generated by Cosmic-rays interacting with the upper atmosphere. The data in figure 6.40, from a study by Autran, Semikh, Munteanu, Serre, Gasiot and Roche,[91] comes from monitoring equipment located in the French Alps; either, 1,700 m under the Fréjus mountain (i.e., all radiation is generated by the device), or on top of the Plateau de Bure at an altitude of 2,552 m (i.e., radiation sources are local and Cosmic).[90] For confidentiality reasons, the data has been scaled by a small constant.

Figure 6.40 shows how the number of bit-flips increased over time (measured in Megabits per hour), for SRAM fabricated using 130 nm, 65 nm and 40 nm processes. The 130 nm and 65 nm measurements were made underground, and the lower rate of bit-flips for the 65 nm process is the result of improved materials selection, that reduced alpha-particle emissions; the 40 nm measurements were made on top of the Plateau de Bure, and show the impact of external radiation sources.

The soft error rate is usually quoted in FITs (Failure in Time), with 1 FIT corresponding to 1 error per 10^9 hours per megabit, or 10^{-15} errors per bit-hour. Consider a system with 4 GB of DRAM (1000 FIT/Mb is a reasonable approximation for commodity memory,[1805] which increases with altitude, being 10 times greater in Denver, Colorado), the system has an MTBF of $1000 \times 10^{-15} \times 4.096 \times 10^9 \times 8 = 3.2 \times 10^{-2}$ hours (around once every 33 hours). Soft errors are a regular occurrence for installations containing hundreds of terabytes of memory.[870]

The Cassini spacecraft experienced an average of 280 single bit memory errors per day[1783] (in two identical flight recorders containing 2.5G of DRAM; also see fig 8.31). The rate of double-bit errors was higher than expected (between 1.5 and 4.5%) because the incoming radiation had enough energy to flip more than one bit.

Uncorrected soft errors place a limit on the maximum number of computing nodes that can be usefully used by one application; at around 50,000 nodes, a system would spend half its time saving checkpoints, and restarting from previous checkpoints after an error occurred.[1562]

Error correcting memory reduces the probability of an uncorrected error by several orders of magnitude, but with modern systems containing terabytes the probability of an error adversely affecting the result remains high.[870] The Cray Blue Waters system at the National Center for Supercomputing Applications experienced 28 uncorrected memory errors (ECC and Chipkill parity hardware checks corrected 722,526 single bit errors, and 309,359 two-bit errors, a 99.995% success rate).[484] Studies[1242] have investigated assigning variables deemed to be critical to a subset of memory that is protected with error correcting hardware, along with various other techniques.[1160]

Calculating the FIT for processors is complicated.[1116]

Redundancy can be used to continue operating after experiencing a hardware fault, e.g., three processors performing the same calculation, and a majority vote used to decide which outputs to accept.[1973] Software only redundancy techniques include having the compiler generate, for each source code sequence, two or more independent machine code sequences[1552] whose computed values are compared at various check points, and replicating computations across multiple cores[1993] (and comparing outputs). The overhead of duplicated execution can be reduced by not replicating those code sequences that are less affected by register bit flips[584] (e.g., the value returned from a bitwise AND that extracts 8 bits from a 32-bit register is 75% less likely to deliver an incorrect result than an operation that depends on all 32 bits). Optimizing for reliability can be traded off against performance,[1278] e.g., ordering register usage such that the average interval between load and last usage is reduced.[1966]

Developers don't have to rely purely on compiler or hardware support, reliability can be improved by using algorithms that are robust in the presence of *faulty* hardware. For instance, the traditional algorithms for two-process mutual exclusion are not fault tolerant; a fault tolerant mutual exclusion algorithm using $2f+1$ variables, where a single fault may occur in up to f variables is available.[1298] Researchers are starting to investigate how best to prevent soft errors corrupting the correct behavior of various algorithms.[256]

Bombarding a system with radiation increases the likelihood of radiation induced bit-flips,[1258] and can be used for testing system robustness.

The impact of level of compiler optimization on a program's susceptibility to bitflips is discussed in section 11.2.2.

Vendor profitability is driving commodity cpu and memory chips towards cheaper and less reliable products, just like household appliances are priced low and have a short expected lifetime.[1700]

A study by Dinaburg[494] found occurrences of bit-flips in domain names appearing within HTTP requests, e.g., a page from the domain `ikamai.net` being requested rather than from `akamai.net` (the 2.10^{-9} bit error rate was thought to occur inside routers and switches). Undetected random hardware errors can be used to redirect a download to another site,[494] e.g., to cause a maliciously modified third-party library to be loaded.

If all the checksums involved in TCP/IP transmission are enabled, the theoretical error rate is 1 in 10^{17} bits; which for 1 billion users visiting Facebook on average once per day and downloading 2M bytes of Javascript per visit, gives an expected bit flip rate of once every 5 days for a single Facebook user.

6.5.1 System availability

A system is only as reliable as its least reliable critical subsystem, and the hardware on which software runs is a critical subsystem that needs to be included in any application reliability analysis; some applications also require a working internet connection, e.g., for database access.

Before cloud computing became a widely available commercial service, companies built their own clustered computer facilities (low usage rates of such systems[884] is what can make cloud providers more cost effective).

The reliability of Internet access to the services provided by other computers is currently not high enough for people to overlook the possibility that failures can occur[182] (see the example in section 10.5.4).

Long-running applications need to be able to recover from hardware failures, if they are to stand a reasonable chance of completing. A process known as *checkpointing* periodically stores the current state of every compute unit, so that when any unit fails, it is possible to restart from the last saved state, rather than restarting from the beginning. A tradeoff has to be made[1969] between frequency of checkpointing, which takes resources away from completing execution of the application but reduces the total amount of lost calculation, and infrequent checkpointing, which diverts less resources but incurs greater losses when a fault is experienced. Calculating the optimum checkpoint interval[423] requires knowing the distribution of node uptimes; see figure 6.41.

The Los Alamos National Laboratory (LANL) has made public, data from 23 different systems installed between 1996 and 2005.[1072] These systems run applications that "... perform long periods (often months) of CPU computation, interrupted every few hours by a few minutes of I/O for check-pointing." Figure 6.41 shows the 10-hour binned data fitted to a zero-truncated negative binomial distribution for systems 2 and 18.

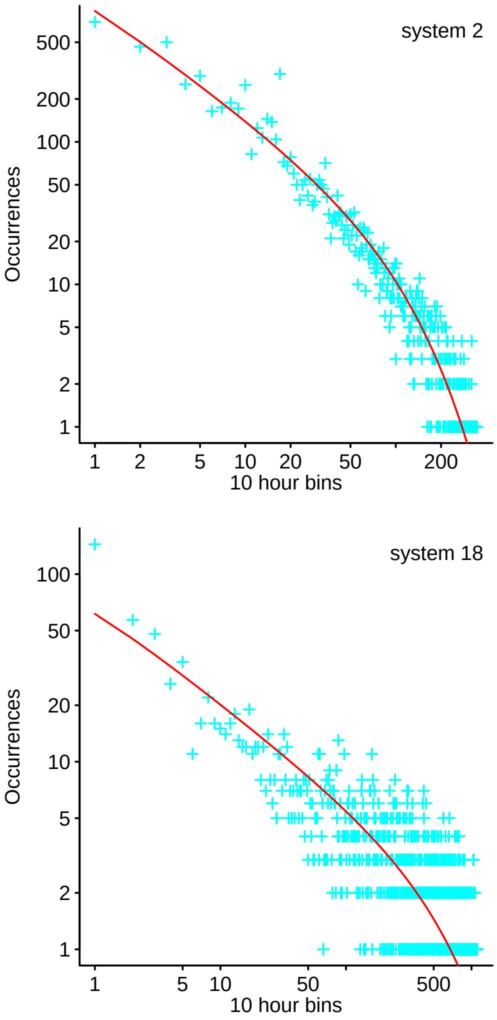


Figure 6.41: For systems 2 and 18, number of uptime intervals, binned into 10 hour intervals, red lines are both fitted negative binomial distributions. Data from Los Alamos National Lab (LANL). Github–Local

Operating systems and many long-running programs sometimes write information about a variety of events to one or more log files. One study[1977] found that around 1 in 30 lines of code in Apache, Postgresql and Squid was logging code; this information was estimated to reduce median diagnosis time by a factor of 1.4 to 3. The information diversity of system event logs tends to increase, with new kinds of information being added, with the writing of older information not being switched off (because it might be useful); log files have been found to contain[1394] large amounts of low value information, more than one entry for the same event, changes caused by software updates, poor or no documentation, and inconsistent information structure within entries.

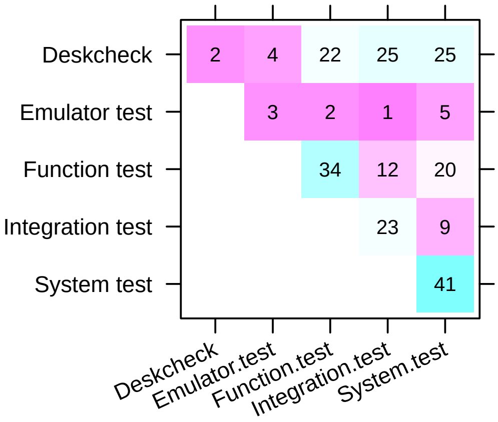
6.6 Checking for intended behavior

The two main methods for checking that code behaves as intended, are: analyzing the source code to work out what it does, and reviewing the behavior of the code during execution (e.g., testing). Almost no data is available on the kinds of mistakes found, and the relative cost-effectiveness of the various techniques used to find them.

So-called *formal proofs* of correctness are essentially a form on N-version programming, with $N = 2$. Two programs are written, with one nominated to be called the specification; one or more tools are used to analyse both programs, checking that their behavior is consistent, and sometimes other properties. Mistakes may exist in the specification program or the non-specification program.[614,660] One study[1302] of a software system that had been formally proved to be correct, found at least two mistakes per thousand lines, remained.

The further along in the development process a mistake is found, the more costly it is likely to be to correct it; possible additional costs include having to modify something created between the introduction of the mistake and its detection, and having to recheck work. This additional cost does not necessarily make it more cost effective to detect problems as early as possible. The relative cost of correcting problems vs. detecting problems, plus practical implementation issues, decide where it is most cost effective to check for mistakes during the development process.

A study by Hribar, Bogovac and Marinčić[854] investigated *Fault Slip Through* by analyzing the development phase where a fault was found compared to where it could have been found. Figure 6.42 shows the number of faults found in various test phases (deskcheck is a form of code review performed by the authors of the code), and where the fault could have been found (as specified on the fault report); also see Antolić.[1848]



	Deskcheck	Emulator.test	Function.test	Integration.test	System.test
Deskcheck	2	4	22	25	25
Emulator test		3	2	1	5
Function test			34	12	20
Integration test				23	9
System test					41

Figure 6.42: Fault slip throughs for a development project at Ericsson; y-axis lists phase when fault could have been detected, x-axis phase when fault was found. Data from Hribar et al.[854] Github–Local

The cost of correcting problems will depend on the cost characteristics of the system containing the software; developing software for a coffee vending machine is likely to be a lot cheaper than for a jet fighter, because of, for instance, the cost of the hardware needed for testing. Data from NASA and the US Department of Defense, on the relative costs of fixing problems discovered during various phases of development are large, because of the very high cost of the hardware running the software systems developed for these organizations.

To reduce time and costs, the checking process may be organized by level of abstraction, starting with basic units of code (or functionality), and progressively encompassing more of the same, e.g., unit testing is performed by individual developers, integration testing checks that multiple components or subsystems work together, and systems testing is performed on the system as a whole.

A study by Nichols, McHale, Sweeney, Snavely and Volkman[1358] investigated the economics of detecting mistakes in system development (the organizations studies all used Team Software Process). One of the organizations studied developed avionics software, which required performing manual reviews and inspections of the high-level design, design and coding phases, followed by testing.

Figure 6.43 shows the reported time taken to correct 7,095 mistakes (for one avionics project), broken down by phase introduced/corrected, against the number of major phases between its introduction and correction (x-axis). Lines are fitted exponentials, with fix times less than 1, 5 and 10-minutes excluded (72% of fixes are recorded as taking less than 10-minutes).

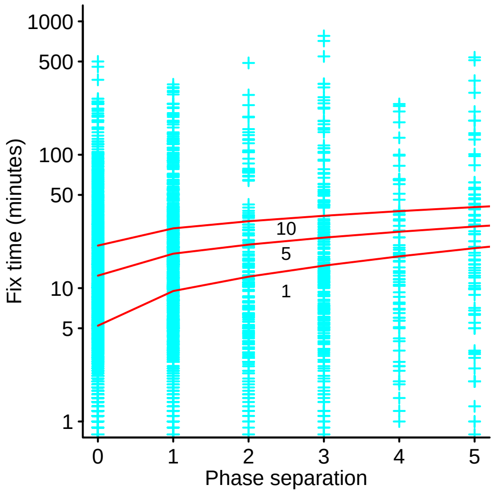


Figure 6.43: Reported time taken to correct 7,095 mistakes (in one project), broken down by phase the mistake was introduced/corrected (y-axis), against number of phases between introduction/correction (x-axis); lines are fitted regression models of the form: $Fix\_time \propto e^{\sqrt{phase\_sep}}$, with fix times less than 1, 5 and 10-minutes excluded. Data from Nichols et al.[1358] Github–Local

Many software systems support a range of optional constructs, and support for these may be selected by build time configuration options. When checking for intended behavior, a decision has to be made on the versions of the system being checked; some systems support so many options, that checking whether all possible configurations can be built requires an unrealistic investment of resources[759] (algorithms for sampling configurations are used[1238]).

6.6.1 Code review

Traditionally a code review (other terms include *code inspection* and *walkthroughs*[623]) has involved one or more people reading another developer's code, and then meeting with the developer to discuss what they have found. These days the term is also applied to reviewing code that has been pushed to a project's version control system, to check whether it is ok to merge the changes into the main branch; with geographically disperse teams, online reviews and commenting have become a necessity.

Review meetings support a variety of functions, including: highlighting of important information between project members (i.e., ensuring that people are kept up to date with what others are doing), and uncovering potential problems before changes becomes more expensive. Detecting issues may not even be the main reason for performing code reviews,[100] keeping teams members abreast of developments and creating an environment of shared ownership of code may be considered more important.

A variety of different code review techniques have been proposed, including: Ad-hoc (no explicit support for reviewers), Checklist (reviewers work from a list of specific questions that are intended to focus attention towards common problems), Scenarios-based (each reviewer takes on a role intended to target a particular class of problems), and Perspective-based reading (reviewers are given more detailed instructions, than they are given in Scenario-based reviews, about how to read the document; see section 13.2 for an analysis). The few experimental comparisons of review techniques have found that the relative performance of the techniques is small compared to individual differences in performance.

The range of knowledge and skills needed to review requirements and design documents may mean that those involved focus on topics that are within their domain of expertise.[526] Many of the techniques used for estimating population size assume that capture sites (i.e., reviews) have equal probabilities of flagging an item; estimates based on data from meetings where reviewers have selectively read documents will be biased; see Github–reliability/eickt1992.R.

Most published results from code review studies have been based on small sample sizes. For instance, Myers,[1322] investigated the coding mistakes detected by 59 professionals, using program testing and code walkthroughs/inspections, for one PL/1 program containing 63 statements and 15 known mistakes; see Github–reliability/myers1978.R. Also, researcher often use issues-found as the metric for evaluating review meetings, in particular potential fault experiences found during code reviews. Issues found is something that is easy to measure, code is readily available, and developers to review it are likely to be more numerous than people with the skills needed to review requirements and design documents (which do not always exist, as such).

Studies where the data is available include:

- Hirao, Ihara, Ueda, Phannachitta and Matsumoto[824] investigated the impact of positive and negative code reviews on patches being merged or abandoned (for Qt and OpenStack). A logistic regression model found that for Qt positive votes were more than twice as influential, on the outcome, as negative votes, while for, OpenStack negative votes were slightly more influential (see Github–reliability/OSS2016.R).
- Porter, Siy, Mockus and Votta[1487] recorded code inspection related data from a commercial project over 18 months (staffed by six dedicated developers, and five developers who also worked on other projects). The best fitting regression model had the number of mistakes found proportional to the log of the number of lines reviewed, and the log of meeting duration; this study is discussed in section 13.2, also see fig 11.33.
- Finifter[595] investigated mistakes found and fault experiences, using manual code review and black box testing, in nine implementations of the same specification. Figure 6.44 shows the number of vulnerabilities found by the two techniques in the nine implementations; some of the difference is due to the variation in the abilities and kinds of mistakes made by different implementers, plus skill differences in using the programming languages.

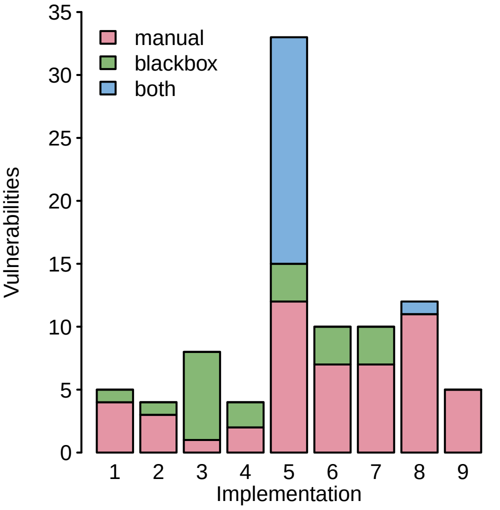


Figure 6.44: Number of vulnerabilities found using blackbox testing, and manual code review of nine implementations of the same specification. Data from Finifter.[595] Github–Local

While there has been a lot of activity applying machine learning to fault prediction, the models have not proved effective outside the data used to build them, or even between different versions of the same project;[2007] see Github–faults/eclipse/eclipse-pred.R. Noisy data is one problem, along with a lack of data on program usage (see section 6.1.3).

During a review, there is an element of chance associated with the issues noticed by individual reviewers, and some issues may only be noticed by reviewers with a particular skill or knowledge. If all reviewers have the same probability, p, of finding a problem, and there are N issues available to be found, by S reviewers, then the expected number of issues found is: $N\left[1-(1-p)^S\right]$.

A study by Nielsen and Landauer[1360] investigated the number of different usability problems discovered, as the number of subjects increased, based on data from 12 studies.

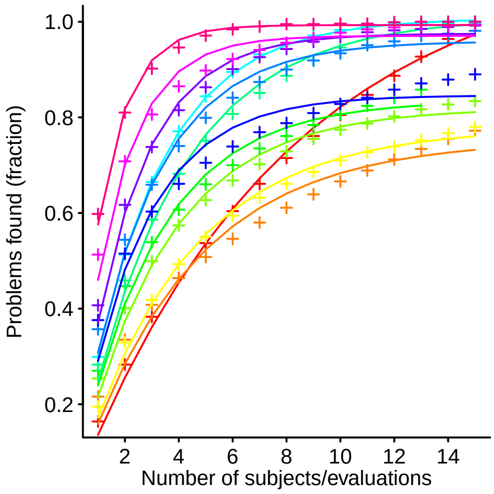


Figure 6.45: Fraction of usability problems found by a given number of subjects/evaluations in 12 system evaluations; lines are fitted regression model for each system. Data extracted from Nielsen et al.[1360] Github–Local

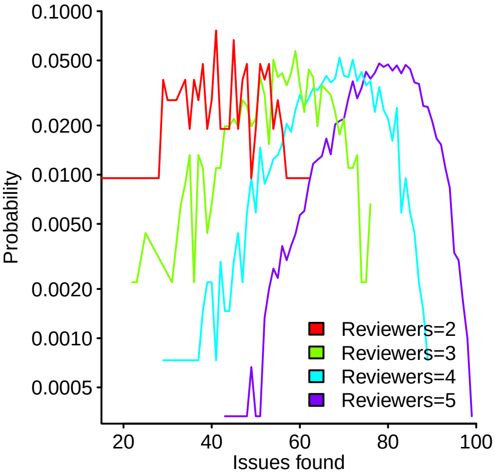


Figure 6.46: Probability (y-axis) of a given number of issues being found (x-axis), by a review group containing a given number of people (colored lines). Data from Lewis.[1111] Github–Local

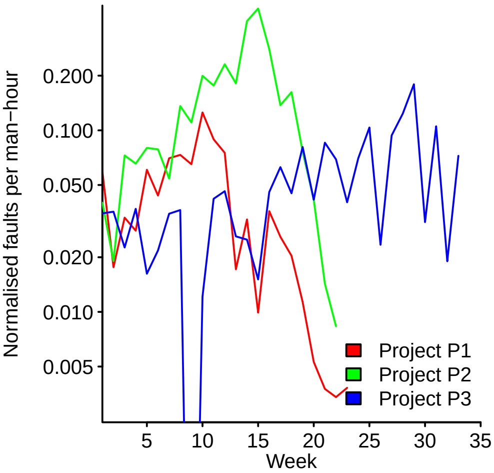


Figure 6.47: Number of faults experienced per unit of testing effort, over a given number of weeks (each normalised to sum to 100). Data from Stikkel.[1760] Github–Local

Figure 6.45 shows how the number of perceived usability problems increased as the number of subjects increased; lines show the regression model fitted by the above equation (both N and p are constants returned by the fitting process).

When the probability of finding a problem varies between reviewers, there can be a wide variation in the number of problems reported by different groupings of individuals.

A study by Lewis[1111] investigated usability problem-discovery rates; the results included a list of the 145 usability problems found by 15 reviewers. How many problems are two of these reviewers likely to find, how many are three likely to find? Figure 6.46 is based on the issues found by every pair, triple (etc, up to five) of reviewers. The mean of the number of issues found increases with review group size, as does the variability of the number found. Half of all issues were only found by one reviewer, and 15% found by two reviewers.

Some coding mistakes occur sufficiently often that it can be worthwhile searching for known patterns. Ideally coding mistakes flagged by a tool are a potential cause of a fault experience (e.g., reading from an uninitialized variable), however the automated analysis performed may not be sophisticated enough to handle all possibilities[1617] (e.g., there may be some uncertainty about whether the variable being read from has been written to), or the usage may simply be suspicious (e.g., use of assignment in the conditional expression of an `if-statement`, when an equality comparison was intended, i.e., the single character = had been typed, instead of the two characters ==). The issue of how developers might respond to false positive warnings is discussed in section 9.1.

A study of one tool[1997] found a strong correlation between mistakes flagged, and faults experienced during testing, and faults reported by customers (after the output of the tool had been cleaned by a company specializing in removing false positive warnings from static analysis tool output).

6.6.2 Testing

The purpose of testing is to gain some level of confidence that software behaves in a way that is likely to be acceptable to customers. For the first release, the behavior may be specified by the implementation team (e.g., when developing a product to be sold to multiple customers), or the customer (e.g., the vendor is interested in meeting the criteria for acceptance, so they get paid). Subsequent releases usually include checks that the behavior is consistent with previous releases.

During testing, a decrease in the number of previously unseen fault experiences, per unit of test effort, is sometimes taken as an indication that the software is becoming more reliable; other reasons for a decrease in new fault experiences is replacement of existing testers by less skilled staff, or repetition of previously used input values. The extent to which reliability improves, as experienced by the customer, depends on the overlap between the input distribution used during testing, and the input distribution provided in real world use.

A study by Stikkel[1760] investigated three industrial development projects. Figure 6.47 shows the number of faults discovered per man-hour of testing, averaged over a week, for these projects (each normalised to sum to 100). The sharp decline in new fault experiences may be due to there being few mistakes remaining, a winding down of investment in the closing weeks of testing (i.e., rerunning the same tests with the same input), or some other behavior.

An example of how the input used for random testing can be unrepresentative of customer input is provided by a study[359] that performed random testing of the Eiffel base library. The functions in this library contain extensive pre/post condition checks, and random testing found twice as many mistakes in these checks as the implementation of the functionality; the opposite of the pattern seen in user fault reports.

To what extent are fault experiences generated by fuzzers representative of faults experienced by users of the software?

A study by Marcozzi, Tang, Donaldson and Cadar[1191] investigated the extent to which fault experiences obtained using automated techniques are representative of the fault experiences encountered by code written by developers. The source code involved in the fixes of 45 reported faults in the LLVM compiler were instrumented to log when the code was executed, and when the condition needed to trigger the fault experience occurred; the following is an example of instrumented code:

```
warn ("Fixing patch reached");
if (Not.isPowerOf2()) {
   if (!(C-> getValue().isPowerOf2()  // Check needed to fix fault
         && Not != C->getValue()))
      {
      warn("Fault possibly triggered");
      }
   else { /* CODE TRANSFORMATION */ } } // Original, unfixed code
```

The instrumented compiler was used to build 309 Debian packages (around 10 million lines of C/C++), producing possibly miscompiled versions of the packages; the build process included running each package's test suite. A package built from miscompiled code may successfully pass its test suite.

A bitwise compare of the program executables generated by the unfixed and fixed compilers was used to detect when different code was generated.

Table 6.3 shows a count, for each fault detector (Human, fuzzing tools, and one formal verifier), of fix locations reached, fix condition triggered, bitwise difference of generated code and failed tests (build tests are not expected to fail). One way of measuring whether there is a difference between faults detected (column 1) in human and automatically generated code is to compare number of fault triggers encountered (column 4).

Detector	Faults	Reached	Triggered	Bitwise-diff	Tests failed
Human	10	1,990	593	56	1
Csmith	10	2,482	1,043	318	0
EMI	10	2,424	948	151	1
Orange	5	293	35	8	0
yarpgen	2	608	257	0	0
Alive	8	1,059	327	172	0

Table 6.3: Fault detector, number of source locations fixed, number of fix locations reached, number of fix condition triggered, number of programs having a bitwise difference of generated code and number of failed tests. Data from Marcozzi et al.[1191]

Comparing the counts for the number of trigger occurrences experienced for each of the 10 fixes in each of the Human, Csmith and EMI detected source mistakes, finds that the differences between the counts is not statistically different across method of detection; see Github–reliability/OOPSLA-compiler.R.

The behavior of some software systems is sufficiently important to some organizations that they are willing to fund the development of a test suite intended to check behavior: cases include:

- the economic benefits of being able to select from multiple hardware vendors is dependent on being able to port existing software to the selected hardware; at a minimum, different compilers must be capable of processing the same source code to produce the same behavior. The US Government funded the development of validation suites for Cobol and Fortran,[1395] and later SQL, POSIX and Ada;[5] a compiler testing service was also established,[10]
- there were a sufficient number of C compiler vendors that several companies were able to build a business supplying test suites for this language, expanding to support C++ when this language started to become popular,[ix]
- the "Write once, run anywhere" design goal for Java required Sun Microsystems to fund the development of a conformance test suite, and to litigate when licensees shipped products that did not conform.[1301, 1930]

While manual tests can be very effective, creating them is very time-consuming[1395] and expensive. Various kinds of automatic test generation are available, including exhaustive testing of short sequences of input[1989] and fuzzing.[1984]

Testing that a program behaves as intended requires knowledge of the intended behavior, for a given input. While some form of automatic input generation is possible, in only a few cases[59] is it possible to automatically predict the expected output from the input,

[ix] A vendor of C/C++ validation suites (selling at around $10,000), once told your author they had over 150 licensees; a non-trivial investment for a compiler vendor.

independently of the software being tested. One form of program behavior is easily detected: abnormal termination, and some forms of fuzz testing use this case as their test criteria (see fig 6.23).

When multiple programs supporting the same functionality are available, it may be possible to use differential testing to compare the outputs produced from a given input (a difference being a strong indicator that one of the systems is behaving incorrectly).[337]

There are ISO Standards that specify methods for measuring conformance to particular standards[889, 890] and requirements for test laboratories.[887] However, very few standards become sufficiently widely used for it to be commercially viable to offer conformance testing services.

Like source code, tests can contain mistakes.[1849]

To what extent do test suites change over time? A study by Marinescu, Hosek and Cadar[1194] measured the statement and branch coverage of six open source programs over time, using the test suite distributed with the program's source. Figure 6.48 shows that for some widely used programs the statement coverage of the test suite did not vary much over five years.

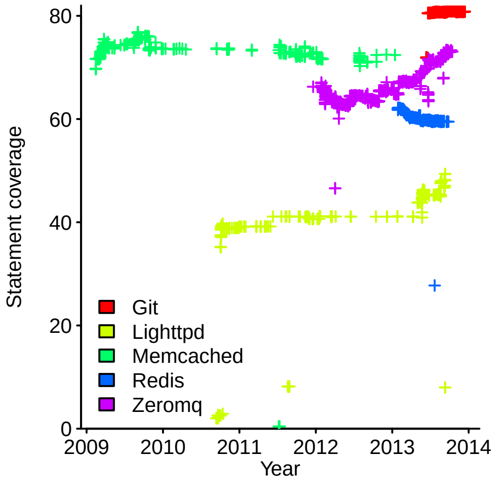


Figure 6.48: Statement coverage achieved by the respective program's test suite (data on the sixth program was not usable). Data from Marinescu et al.[1194] Github–Local

One study[1979] found no correlation between the growth of a project and its test code; see Github–time-series/argouml_complete.R.

The application build process may require the selection of a consistent set of configuration options. The Linux 2.6.33.3 kernel supports 6,918 configuration options, giving over $10^{23,563}$ option combinations. One study[1127] using a random sample of 1,000,000 different option combinations failed to find any that were valid according to the variability model; a random sampling of the more than $10^{1,377}$ possible option combinations supported by OpenSSL found that 3% were considered valid. Various techniques have been proposed for obtaining a sample of valid configuration options,[953] which might then be used for testing different builds of an application, or analyzing the source code.[1890]

Regular expressions are included within the syntax of some languages (e.g., awk and SNOBOL 4[735]), while in others they are supported by the standard library.[325] A given regular expression may match (or fail to match) different character sequences in different languages[436] (e.g., support different escape sequences, and different disambiguation policies; PCRE based libraries use a leftmost-greedy disambiguation, while POSIX based libraries use the leftmost-longest match[179]).

A study by Wang and Stolee[1901] investigated how well 1,225 Java projects tested the regular expressions appearing in calls to system libraries (such as `java.lang.String.matches`); there were 18,426 call sites. The regular expression methods were instrumented to obtain the regular expression and the input strings passed as arguments. When running the associated project test suite 3,096 (16.8%) of call sites were evaluated; method argument logging during test execution obtained 15,096 regular expressions and 899,804 test input strings (at some call sites, regular expressions were created at runtime).

A regular expression can be represented as a deterministic finite state automata (DFA), with nodes denoting states and each edge denoting a basic subcomponent of the regular expression. Coverage testing of a regular expression involves counting the number of nodes and edges visited by the test input.

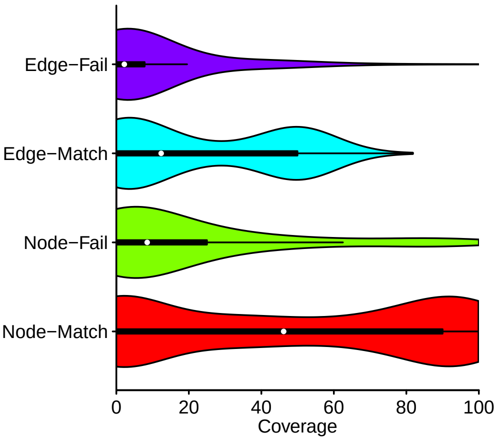


Figure 6.49: Violin plots of percentage of regular expression components having a given coverage, (measured using the nodes and edges of the DFA representation of the regular expression, broken down by the match failing/succeeding) for 15,096 regular expressions, when passed the corresponding project test input strings. Data kindly provided by Wang.[1901] Github–Local

Figure 6.49 shows a violin plot of the percentage of regular expression components having a given coverage. The nodes and edges of the DFA representation of each of the 15,096 regular expressions are the components measured, using the corresponding test input strings for each regex; coverage if measured for both matching and failing inputs.

6.6.2.1 Creating tests

Traditionally tests were written by people employed to test software; some companies have Q/A (quality assurance) departments. The tests developers write to check their code may become part of the systems formal test suite; there are development methodologies in which include testing is a major component of implementation, e.g., test driven development.

Automated test generation can reduce the time and costs associated with testing software. A metric often used by researchers for evaluating test generation tools is the number of fault experiences produced by the generated tests (i.e., one of the factors involved in gaining confidence that program behavior is likely to be acceptable).

Automated test generation techniques include (there is insufficient evidence to evaluate the extent to which an automatic technique is the most cost effective to use in a particular development):

- random modification of existing tests: so-called *fuzzing* makes random changes to existing test inputs, and little user input is required to test for one particular kind of fault experience, abnormal termination (see fig 6.23). Some tools use a fuzzing selection strategy intended to maximise the likelihood of generating a file that causes a crash fault, e.g., CERT's BFF uses a search strategy which gives greater weight to files that have previously produced faults[851] (i.e., it is a biased random process),
- source code directed random generation: this involves a fitness function, such as number of statements covered or branches executed.

  A study by Salahirad, Almulla and Gay[1608] investigated the ability of eight fitness functions, implemented in the EvoSuite tool, to generate tests that produced fault experiences for 516 known coding mistakes; a test generation budget of two and ten-minutes per mistake was allocated on the system used. The branch coverage fitness function was found to generate tests that produced the most fault experiences,
- input distribution directed random generation: the generation process uses information about the characteristics of the expected input (e.g., the probability of an item appearing, or appearing in sequence with other items) to generate tests having the same input characteristics.

  A study by Pavese, Soremekun, Havrikov, Grunske and Zeller[1435] used the measured characteristics of input tests to create a probabilistic grammar that generated tests having either the same distribution or were uncommon inputs (created by inverting the measured input item probabilities).

Automated test generation techniques are used to find vulnerabilities by those seeking to hijack computer systems for their own purposes. To counter this threat, tools and techniques have been created to make automatic test generation less cost effective.[948]

A program's input may be include measurements of a set of different items (e.g., the time of day, the temperature and humidity), and within the code there may be an interaction between these different items (these items are sometimes called *factors*). A coding mistake may only be a source of a fault experience when two different items each take a particular range of values, and not when just one of the items is in this range.

Combinatorial testing involves selecting patterns of input that are intended to detect situations where a fault experience is dependent on a combination of different item input values. The generation of the change patterns used in combinatorial testing can be very similar to those used in the design of experiments, and the same techniques can be used to help minimise the number of test cases; see section 13.2.5.

A study by Kuhn, Kacker and Lei[1040] investigated the percentage of fault experiences likely to require a given number of factors, in some combination. Figure 6.50 shows the cumulative growth in the percentage of known faults experienced, for tests involving combinations of a given number of factors (x-axis).

A study by Czerwonka[422] investigated the statement and branch coverage achieved, in four Microsoft Windows utilities, by combinatorial tests. The tests involved values for a single factor, and interaction between factors (from two to five factors). The results found that most of the variance in the measurements of branch and statement coverage could be explained by models fitted using the *log* of the number of combination of factors and the *log* of the number of tests; see Github–reliability/Coverage-Combin.R.

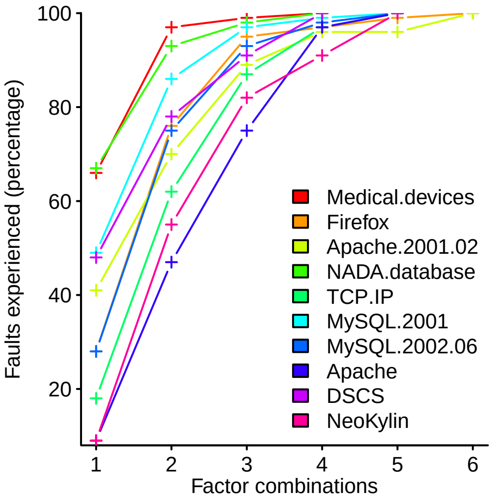


Figure 6.50: Percentage of known faults experienced for tests involving a given number of combinations of factors (x-axis), for ten programs. Data from Kuhn et al.[1040] Github–Local

6.6.2.2 Beta testing

The input profiles generated by the users of a software system may be very different from those envisaged by the developers who tested the software. Beta testing is a way of discovering problems with software (e.g., coding mistakes and incomplete requirements), when processing the input profiles of the intended users. The number of problems found during beta testing, compared to internal testing provides feedback on the relevance of the usage profile that drives the test process.[1188]

Beta testing is also a form of customer engagement.

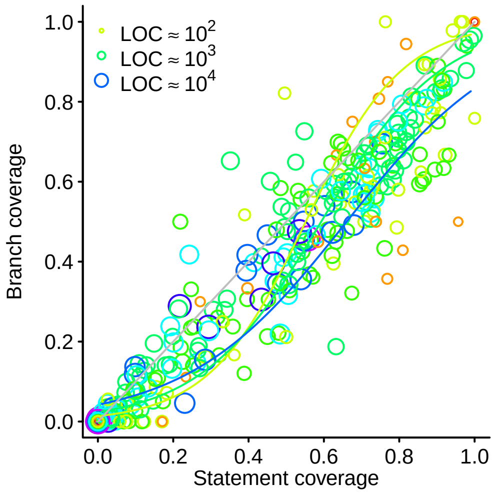


Figure 6.51: Statement coverage against branch coverage for 300 or so Java projects; colored lines are fitted regression models for three program sizes (see legend), equal value line in grey. Data from Gopinath et al.[704] Github–Local

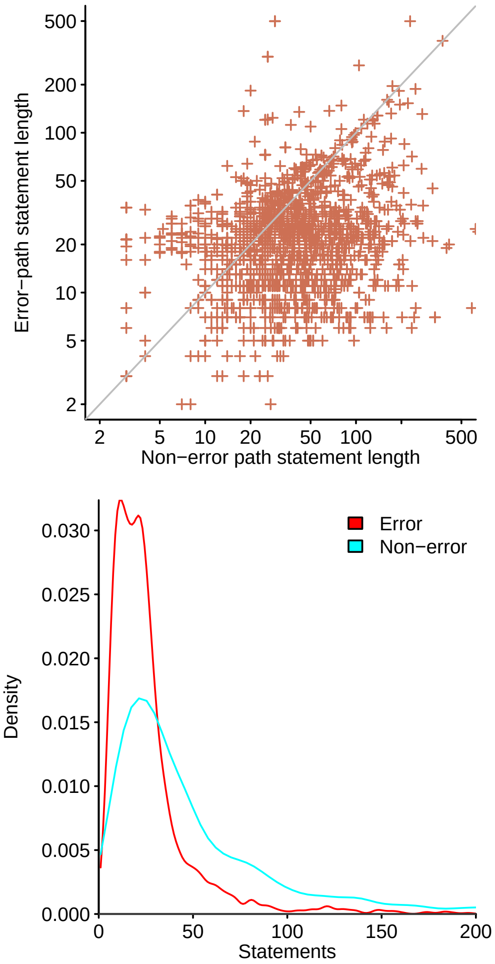


Figure 6.52: Number of statements executed along error and non-error paths within a function (top), and density plots of the number of statements along error and non-error paths. Data kindly provided by Kang.[962] Github–Local

6.6.2.3 Estimating test effectiveness

Does the project test process provide a reliable estimate, to the desired level of confidence, that the software is likely to be acceptable to the customer?

A necessary requirement for checking the behavior of code is executing it, every statement not executed is untested. Various so called *coverage* criteria are used, for instance percentage of program methods or statements executed by the test process (the coverage achieved by a test suite is likely to vary between platforms, different application configurations,[1523] and even the compiler used[643]).

The conditional expression in `if-statements` controls whether a block of statements is executed, or not. Branch coverage simply counts the number of branches executed, e.g., one branch for each of the true and false arms of an `if-statement`. More thorough coverage criteria measure the coverage of the decisions involved in the conditional expression, which can be an involved process; an expression may involve decisions conditions (e.g., `x && (y || z)`), with each subcondition derived from a different requirement. Modified Condition and Decision Coverage (MC/DC)[344] is one measure of coverage of the combination of possible decisions that may be involved in the evaluation of a conditional expression. For this example, the MC/DC requirements are met by `x`, `y` and `z` (assumed to take the values `T` or `F`) taking the values: `TFF`, `TTF`, `TFT`, and `FTF`, respective.

The probability of tests written to the MC/DC requirements detecting an incorrect condition is always at least 93.75%;[345] see Github–reliability/MCDC_FP.R.

One weakness of MC/DC is its dependence on the way conditions are expressed in the code. In the previous example, assigning a subexpression to a variable: `a=(y || z);`, simplifies the original expression to: `x && a`, and reduces the number of combinations of distinct values that need to be tested to achieve 100% MC/DC coverage.[x] One study[652] was able to restructure code to achieve 100% MC/DC coverage using 50% fewer tests than the non-restructured code (in some cases even fewer tests were needed). Achieving MC/DC coverage is often a requirement for software used within safety critical applications.

A study by Inozemtseva and Holmes[879] investigated test coverage of five very large Java programs. The results showed a consistent relationship between percentage statement coverage, *sc*, and percentage branch coverage, *bc* (i.e., $bc \propto sc^{1.2}$), and percentage modified condition coverage, *mc* (i.e., $mc \propto sc^{1.7}$); see Github–reliability/coverage_icse-2014.R.

The most common use of branches is as a component of the conditional expression in an `if-statement`, which decides whether to execute the statements in the enclosed compound statement. Most compound statements contain a small number of statements,[919] so a close connection between branch and statement coverage is to be expected.

A study by Gopinath, Jensen and Groce[704] investigated the characteristics of coverage metrics for the test suites of 1,023 Java projects. Figure 6.51 shows the fraction of statement coverage against branch coverage; each circle is data from one project. The various lines are fitted regression models, which contain a non-simple interaction between coverage and $\log(KLOC)$.

In figure 6.51, why does branch coverage tend to grow more slowly than statement coverage? Combining the findings from the following two studies suggest a reason:

- A study by Kang, Ray and Jana[962] investigated the number of statements encountered along the execution paths, within a function, executed after a call to a function that could return an error, i.e., the error and non-error paths. Figure 6.52, lower plot, shows that non-error paths often contain more statements than the error paths; in the upper plot most of the points are below the line of equal statement-path length (i.e., there are a greater number of longer non-error paths).
- A study by Čaušević, Shukla, Punnekkat and Sundmark[1841] found that developers wrote almost twice as many positive tests as negative tests,[xi] for the problem studied; see Github–reliability/3276-TDD.R. This behavior may be confirmation bias; see section 2.2.1.

[x] Some tools track variables appearing in conditionals that have previously been assigned expressions whose evaluation involved equality or relational operators.

[xi] Sometimes known as error and non-error tests.

If a test suite contains more positive than negative tests, and positive tests involve more statements than negative tests, then statement coverage would be expected to grow faster than branch coverage.

A basic-block is a sequence of code that has one entry point and one exit point (a function call would be an exit point, as would any form of goto statement). In figure 6.53, the fitted regression line shows a linear relationship between basic-block coverage and decision coverage, i.e., the expected relationship (grey line shows *Decision* = *Block*).

A study by McAllister and Vouk[1212] investigated the coverage of 20 implementations of the same specification. Two sets of random tests were generated using different selection criteria, along with 796 tests designed to provide full functional coverage of the specification. Figure 6.54 shows the fraction of basic-blocks covered as the number of tests increases, for the 20 implementations (sharing the same color), and three sets of tests (the different colors); the lines are fitted regression models of the form: $coverage_{BB} = a \times (1 - b \times \log(tests)^c)$, where: a, b and c are constants (c is between -0.35 and -1.7, for this specification).

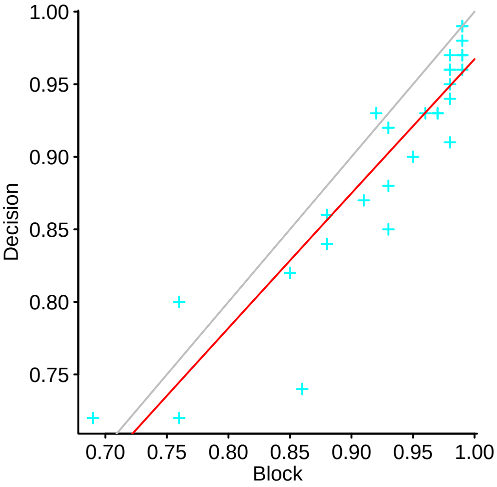


Figure 6.53: Basic-block coverage against branch coverage for a 35 KLOC program; lines are a regression fit (red) and *Decision* = *Block* (grey). Data from Gokhale et al.[685] Github–Local

Another technique for estimating the effectiveness of a test suite, in detecting coding mistakes, is to introduce known mistakes into the source and measure the percentage detected by the test suite, i.e., the mistake produces a change of behavior that is detected by the test process; the modified source is known as a *mutant*. For this technique to be effective, the characteristics of the mutations have to match the characteristics of the mistakes made by developers; existing mutant generating techniques don't appear to produce coding mistakes that mimic the characteristics of developer coding mistakes.[702, 705] A test suite that kills a high percentage of mutants (what self-respecting developer would ever be happy just detecting mutants?) is considered to be more effective than one killing a lesser percentage.

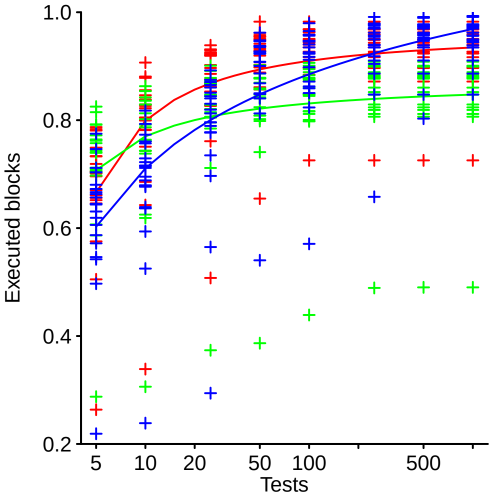


Figure 6.54: Fraction of basic-blocks executed by a given number of tests, for 20 implementations using three test suites. . Data from McAllister et al.[1212] Github–Local

Figure 6.55 shows statement coverage against percentage of mutants killed. The various lines are fitted regression models, which contain a non-simple interaction between coverage and $\log(KLOC)$.

A test suite's mutation score converges to a maximum value, as the number of mutants used increases. For programs containing fewer than 16K executable statements, E, the number of mutants needed has been found to grow no faster than $O(E^{0.25})$,[1988] a worst case confidence interval for the error in the mutation score can be calculated.[703]

6.6.3 Cost of testing

Testing costs include the cost of creating the tests, subsequent maintenance costs (e.g., updating them to reflect changes in program behavior), and the cost of running the tests.

For some kinds of test creation, automatic test generation tools may be more cost effective than human written tests;[328] see Github–reliability/1706-01636a.R.

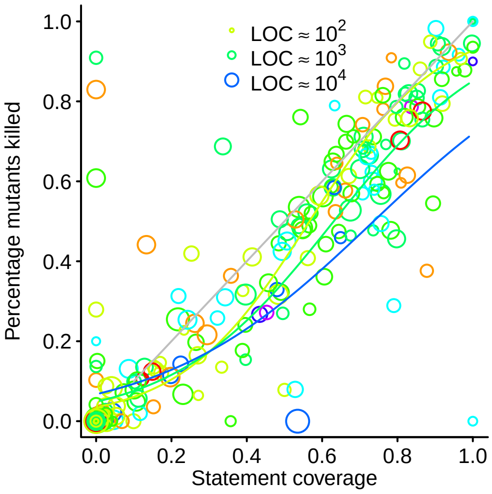


Figure 6.55: Statement coverage against mutants killed for 300 or so Java projects; colored lines are fitted regression models for three program sizes, equal value line in grey. Data from Gopinath et al.[704] Github–Local

The higher the cost of performing system testing, the fewer opportunities there are likely to be to check software at the system level. Figure 6.56 shows the relationship between the unit cost of a missile (being developed for the US military), and the number of development test flights made.

When does it become cost effective to stop testing software? Some of the factors involved in stopping conditions are discussed in section 13.2.4. Studies[323, 1968] have analysed stopping rules for testing after a given amount of time, based on number of faults experienced.

A study by Do, Mirarab, Tahvildari and Rothermel[497] investigated test case prioritization, and attempted to model the costs involved; multiple versions of five programs (from 3 to 15 versions) and their respective regression suites were used as the benchmark. The performance of six test prioritization algorithms were compared, based on the number of seeded coding mistakes detected when test resource usage was limited (reductions of 25, 50 and 75% were used). The one consistent finding was that the number of faults experienced decreased as the amount of test resource used decreased, there were interactions between program version and test prioritization algorithm; see Github–reliability/fse-08.R.

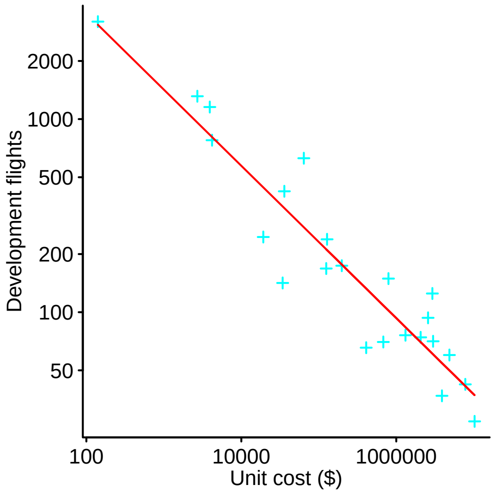


Figure 6.56: Unit cost of a missile, developed for the US military, against the number of development test flights carried out, with fitted power law. Data extracted from Augustine.[85] Github–Local

Chapter 7

Source code

7.1 Introduction

Source code is the primary deliverable product for software development. The purpose of studying source code is to find ways of reducing the resources needed to create and maintain it, and resources such as the developer cognitive effort needed to process code.

The limiting resource during software development is the experience and cognitive effort deliverable by the people involved; the number of people involved may be limited by the funding available, and their individual experience and cognitive firepower is limited to those people that could be recruited.

A study by Ikutani, Kubo, Nishida, Hata, Matsumoto, Ikeda and Nishimoto[876] investigated the neural basis for programming expertise. Subjects (10 top ranking, 10 middle ranking, 10 novices, on the AtCoder competitive programming contest website) categorized 72 Java code snippets into one of four categories (maths, search, sort, string); each snippet was presented three times, for a total of 216 categorization tasks per subject. The tasks were performed while each subject was in an fMRI scanner. A searchlight analysis[554] of the fMRI data was used to locate brain areas having higher levels of activity; figure 7.1 is a composite image of all active locations found over all 30 subjects.[i]

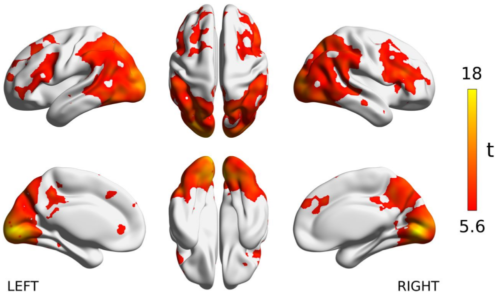


Figure 7.1: Composite image of brain areas active when 30 subjects categorized Java code snippets; colored scale based on t-contrast for source code presentation in a GLM model of the MRI signals. Image from Ikutani et al.[876]

Building a software system involves arranging source code in a way that causes the desired narrative to emerge, as the program is executed, when processing user inputs.

Source code is a form of communication, written in a programming language, whose purpose is often to communicate a sequence of actions, or required results[ii] to a computer, and sometimes a person.

There are many technical similarities between programming languages and human languages, however, the ways in which they are used to communicate are very different (differences are visible in medical imaging of brain activity[609]). While there has been a lot of research investigating the activities involved in processing written representations of human language[434] (i.e., prose[iii]; see section 2.3), there have been few studies of the activities involved in the processing source code by people. For instance, eye-tracking is commonly used to study reading prose, but is only just starting to be used to study reading code; see fig 2.17. This chapter draws on findings from the study of prose (see section 2.3.1), highlighting possible patterns of behavior that might apply to source code.

The influential work of Noam Chomsky[349] led to widespread studying of language based on the products of language (e.g., words and sentences) abstracted away from the context in which they are used. The cognitive linguistics[1531] approach is based around how people use language, with context taken into account (e.g., intentions and social normals, as in the work of Grice[731]). This chapter takes a cognitive linguistics approach to source code, including:

[i] The color scale is a measure (using t-contrasts) of the explanatory power of source code presentation in a GLM model of the recorded MRI signals.

[ii] In a declarative language, such as SQL, the intended result is specified (e.g., return information matching the specified conditions), while code written in an imperative language specifies actions to be performed (with the author being responsible for sequencing the actions to produce an intended result).

[iii] English prose has been the focus of most studies, with prose written in other languages not yet so extensively studied.

- human language use is a joint activity, based on common ground between speaker and listener[iv]. What a speaker says is the evidence used by a listener to create an interpretation of the speaker's intended meaning; a conversation involves cooperation and coordinating activities.

  Grice's maxims[731] provide a model of human communication, these are underpinned by speaker aims and listener assumptions. Perhaps the most important aspect of human language communication is the assumption of relevance[361, 1725] (and even optimal relevance). A speaker says something because they believe it is worth the effort, with both speaker and listener assuming that what is said is relevant to the listener, and worth the listener investing effort to understand,

- communicating with a computer, using source code, is a take-it or leave-it transaction, what the speaker *said* is treated as definitive by the listener, intent is not part of the process. There is no cooperative activity, and the only coordination involves the speaker modifying what has been *said* until the desired listener response is achieved.

  Human readers of source code may attempt to extract an intent behind what has been written.

  Creating a program requires explaining, as code, everything that is needed. The implementation language may support implicit behavior (e.g., casting operands to a common type), or be based on a specific model of the world (e.g., domain specific languages). Commonly occurring narrative subplots may be available as library functions.

Experienced developers are aware of the one-sided nature of communicating with a computer, and have learned a repertoire of techniques for adjusting their approach to communication; novices have to learn how to communicate with a listener who is not aware of their intent. One aspect of learning to program is learning to communicate with an object that will not make any attempt to adapt to the speaker; patterns have been found in the ways novices misunderstanding the behavior of code.[1438]

Source code is used to build programs and libraries (or packages). There are a variety of different kinds of text that might be referred to as *source code*, e.g., text that is compiled to produce an executable program, text that is used to direct the process of building programs and system distributions (such as Makefiles and shell scripts), text that resembles prose in configuration files, and README files[875] containing installation and usage examples.

A study by Pfeiffer[1459] investigated the kinds of files contained in 23,715 Github repositories. Files were classified into four high-level categories (i.e., code, data, documentation, and other), and 19 lower level categories (e.g., script, binary code, build code, video, font, legalese). Figure 7.2 shows the fraction of files in each high-level category within repositories containing a given total number of files, averaged over repositories having the given total numbers of files (the data was smoothed using a rolling mean, with a window width of three).

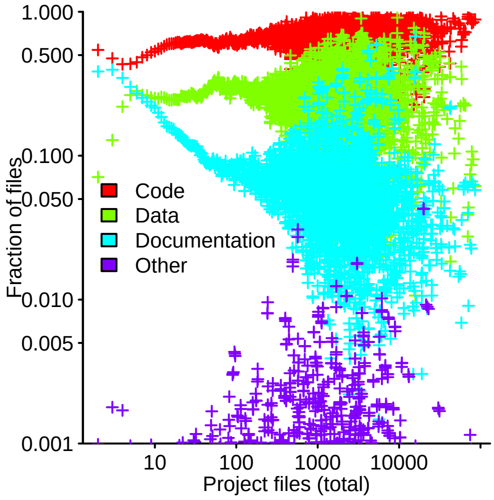


Figure 7.2: Fraction of files in high-level categories for 23,715 repositories containing a given number of files (averaged over all repositories containing a given number of files). Data from Pfeiffer.[1459] Github–Local

Source code is the outcome of choices made in response to implementation requirements, the culturally derived beliefs and experiences of the developers writing the code (coupled with their desire for short-term gratification[622]), available resources, and the functionality provided by the development environment.

In some application domains, effective ways of organizing implementation components have become widely known. For instance, in the compiler writing domain, the production-quality compiler-compiler project[1107] created a way of organizing an optimizing compiler, as a sequence of passes over a tree, that has been widely used ever since.[v]

High-level implementation plans have to be broken down into smaller components, and so on down, until the developer recognises how a solution that can be directly composed using code.

Developers have a collection of beliefs, and mental models, about the semantics of the programming languages they use, as well as a collection of techniques they have become practiced at, and accustomed to using.

The low-level patterns of usage found in source code arise from the many coding choices made in response to immediate algorithmic and implementation requirements. For instance, implementing the $3n+1$ problem requires testing whether n is odd or even. In one study,[1857] the expressions used for this test included: `n % 2` and `n & 1` (89% and 8% of

[iv] "vaj laH: pejatlh lion ghaH yajbe' maH." Wittgenstein.

[v] Recent research has been investigating subcomponent sequencing selected machine learning, on a per-compilation basis.[79]

uses respectively), along with less common sequences such as: `(n >> 1) << 1 ==n` and `(n/2)*2 ==n`; around 50% of the conditions returned a non-zero value (i.e., true), when *n* is even. Over 95% of the expressions appeared as the condition of an `if-statement`, with most of the others appearing as the first operand of a ternary operator, e.g., `n=(n%2)? 3*n+1 :n/2`.

There are a huge variety of different ways of implementing the same functionality (see fig 5.23), and neutral variants of existing programs can be created[772] (i.e., small changes that do not affect the external behavior); the style of some developers source code is sufficiently different from other developers that it can be used to distinguish them from other developers.[285]

If, after executing the two statements: `x=1;y=x+2;`, the statement: `x=3;`, is executed, is the value of `y` now 5? In many programming languages the value of `y` remains unchanged by the second assignment to `x`, but in reactive programming languages[114] (found in spreadsheets) statements can express a relationship, not a one time assignment of a value (novice developers have been found to give a wide variety of interpretations to the assignment operator[219]).

While spreadsheets support the specification of formula and relationships between named memory locations, they have not traditionally been treated as part of software engineering. One reason has been the lack of large samples of usage to study; legal proceedings against large companies are helping to change this situation.[807]

Acquiring an understanding of the behavior of a program, by reading its source code, is not an end in itself; one reason for making an investment to acquire this understanding, is to be able to predict a program's behavior sufficiently well to be able to change it. By reading source code, developers acquire beliefs about it, which are a means to an end; *understanding a program* is a continuum, not a yes/no state.

The complexity and inter-connectedness found in software systems can also be found in non-software systems. A study by Braha and Bar-Yam[235] investigated the network connections in a 16-story hospital, pharmaceutical facilities design, a General Motors vehicle design facility and Linux. Table 7.1 lists the values of various properties of the networks and from a component network perspective software looks middle of the road.

	Nodes	**Edges**	**Average path length**	**Clustering coeff**	**Degree**	**Density**
Hospital	889.00	8178.00	3.12	0.11	18.40	0.02
Pharmaceutical	582.00	3689.00	2.63	0.12	12.68	0.02
Software	466.00	1245.00	3.70	0.14	5.34	0.01
Vehicles	120.00	417.00	2.88	0.13	6.95	0.06

Table 7.1: Values of various attributes of the communication network graphs for various organizations. Data from Braha et al.[235] Github–Local

So-called *end-user* programming involves non-developers writing code to perform simple tasks. Trigger-Action-Programs, written in languages such as IFTTT (If This Then That), can be created by users to control IoT devices (e.g., smart speakers);[1256] spreadsheets[807] are used by a wide range of non-developers; and, Scratch[819] is used for teaching children.

Are the factors driving non-developer written applications sufficiently different from the factors driving developer written applications, that the characteristics of the code written is noticeably different?

This question is outside the scope of this book, which focuses on professional developers.

7.1.1 Quantity of source

The size of software systems is sometimes used to obtain an estimate of the cost of its production, the time taken to implement it, and number of mistakes it contains (see fig 5.1 and fig 5.3). Given the large number of variables in the production process (e.g., the large variation in the quantity of code written by different developers to implement the same functionality, see fig 5.23), size only has any likelihood of good enough accuracy for comparisons within the same project team working on the system.

What is the size distribution of software systems, and to what extent do the characteristics of subcomponents vary with size?

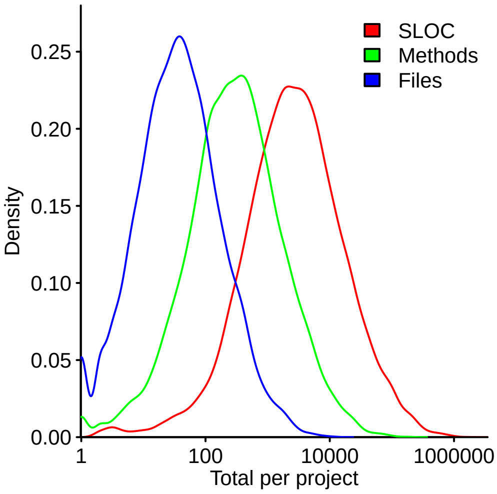


Figure 7.3: Number of source files, methods, and lines of code within methods, contained in each of 13,103 Java projects; lines are kernel density plots. Data kindly provided by Landman.[1065] Github–Local

The size of a software system is often measured by lines of source code. While building a program from source invariably involves a variety of configuration files (e.g., makefiles), such files are not always categorized as source files; counts may specify the kinds of file counted, e.g., those having a given file suffix. Figure 7.3 shows the number of source files, methods and SLOC (within each method) for 13,103 Java projects (y-axis shows density, rather than a count).

There is sufficient consistency in source code layout for lines to be treated as a good enough unit of measure. In many languages a *method* (also known as a *function*, *procedure* or *subroutine*) is the smallest self-contained unit of source code, with larger units of measurement including classes and files. These characteristics are interchangeable in the sense that, when the value of one of them is known, an order of magnitude approximation of the other values can be calculated.

to	SLOC	Methods	Classes	Files
SLOC		$13^{+20}_{-8} \times Methods^{0.97\pm0.05}$	$50^{+100}_{-30} \times Classes^{1.02\pm0.08}$	$64^{+300}_{-50} \times Files^{1.04\pm0.13}$
Methods	$0.11^{+0.2}_{-0.07} \times SLOC^{0.98\pm0.04}$		$4.5^{+10}_{-3} \times Classes^{1.03\pm0.08}$	$7^{+20}_{-5} \times Files^{1.05\pm0.11}$
Classes	$0.054^{+0.07}_{-0.04} \times SLOC^{0.86\pm0.03}$	$0.42^{+0.6}_{-0.3} \times Methods^{0.86\pm0.04}$		
Files	$0.051^{+0.1}_{-0.04} \times SLOC^{0.84\pm0.08}$	$0.23^{+0.4}_{-0.2} \times Methods^{0.89\pm0.07}$		

Table 7.2: Equations that map between total number of Java source constructs in a software system (fitted using quantile regression, the bounds are derived from 95% and 5% quantile regression models). Data from Lopes et al,[1146] and the Files data is from Landman et al.[1065] Github–Local

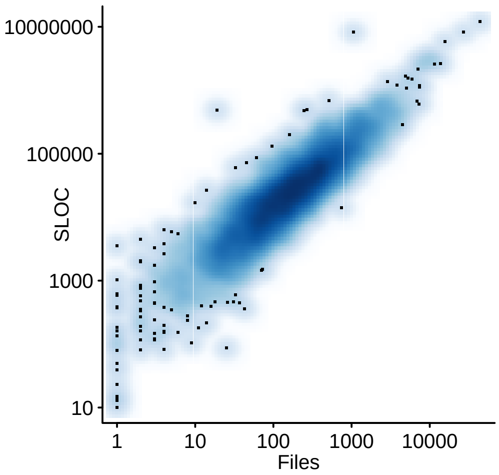


Figure 7.4: Number of files and lines of code in 3,782 projects hosted on Sourceforge. Data from Herraiz.[1785] Github–Local

Table 7.2 shows the equations obtained by fitting quantile regression models to measurements of Java systems containing 10 or more methods (the study by Lopes and Ossher[1146] counted SLOC in the classes of 27,063 Java systems, and Landman, Serebrenik, Bouwers and Vinju[1065] counted SLOC in the methods of 12,628 Java systems). The listed equation uses the median, with uncertainty bounds derived from fitting the 95% and 5% quantiles. More accurate models can be used when information on more characteristics is available, e.g., $SLOC = e^{1.98} Methods^{1.12} Files^{-0.13}$, and $SLOC = e^{1.78} MethodsInClasses^{0.8} Classes^{-0.2}$.

To what extent do the size characteristics of source code written in other languages follow the patterns seen in Java (i.e., power law with an exponent close to one)? The pattern seen in figure 7.14 shows that different size characteristics occur in at least one other language.

A study by Herraiz[1785] measured the number of files and SLOC in a snapshot of 3,781 projects hosted on SourceForge (as of June 2006, having at least three developers and one year of history). Figure 7.4 shows number of files against SLOC, as a density plot; the fitted quantile regression model, with 95% bounds is: $SLOC = 167^{+563}_{-134} \times Files^{0.98\pm0.09}$.

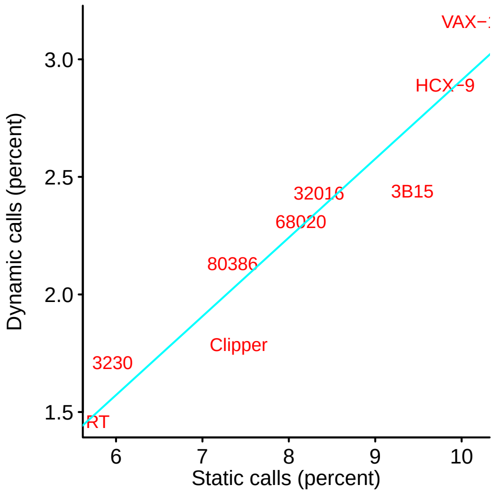


Figure 7.5: Percentage of call instructions contained in code generated from the same C source, against call execution percentage for various processors; grey line is fitted regression model. Data from Davidson et al.[433] Github–Local

Sometimes the only information available about a program is contained in its executable form. Studies[1775] have built models that estimate the length of the original source from the compiled machine code (the language in which the source is written can have a large impact on machine code characteristics[282]).

The relationship between static and dynamic counts of machine code, compiled from the same source, can be noticeably affected by processor characteristics and the compiler used. Figure 7.5 shows static and dynamic percentage of call instruction opcodes, in the compiled form of various C programs targeting various processors.

The quantity of source has been found to be an approximate predictor of compile time. One study[1265] of Ada compilers found that compile time was proportional to the square root of the number of tokens; see Github–sourcecode/ADA177652.R. A study by Auler and Borin[86] investigated the time taken by a JIT compiler to generate code for functions in the SPEC CPU2006 benchmark. Figure 7.6 shows the time taken to generate machine code for 71,200 functions, containing a given number of LLVM instructions (an intermediate code). The two trends in the data are: compile time not depending on function size and compile time increasing linearly (once functions contain more than 100 instructions).

7.1.2 Experiments

Human experiments are the primary technique for unravelling developer performance interactions with source code. Usage patterns in source code are the emergent outcome of developer behavior, application requirements and the characteristics of the development environment. Focusing on common usage patterns can be an efficient use of research resources.

Obtaining reliable experimental results requires controlling all the variables likely to affect the outcome. Asking subjects to solve variations of a specific simple question is one method of excluding extraneous factors. This is a bottom up approach to understanding behavior.

Subjects will vary in ability and questions will vary in difficulty; item response theory can be used to model this kind of performance data.

A study by Chapman, Wang and Stolee[324] investigated the accuracy with which 180 workers on Mechanical Turk (who had to correctly answer 4-5 questions about regular expressions before being accepted) matched regular expressions against particular character sequences (and also constructing a character sequence to match a given regular expression). A set of 41 equivalent regex pairs (equivalent in that the same character sequences were matched by different regexs) was used to construct 60 problems, with 10 randomly selected for each worker to answer.

Figure 7.7 shows the probability that a worker with a given ability (x-axis) will correctly answer a particular problem (numbered colored lines).[vi] Unpicking the various ability/difficulty response patterns requires further work.

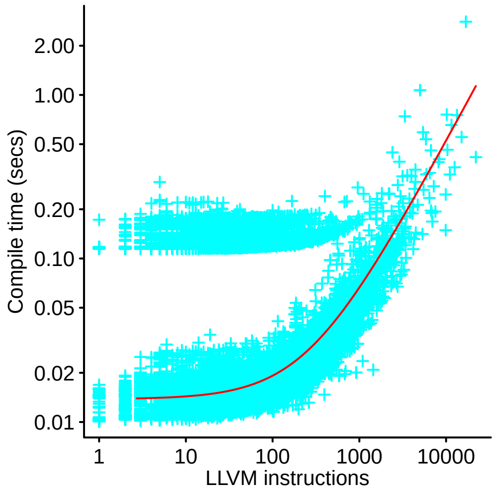


Figure 7.6: Time to compile, using -O3 optimization, each of 71,200 function (in the SPEC benchmark) containing a given number of LLVM instructions; line shows fitted regression model for one trend in the data. Data kindly provided by Auler.[86] Github–Local

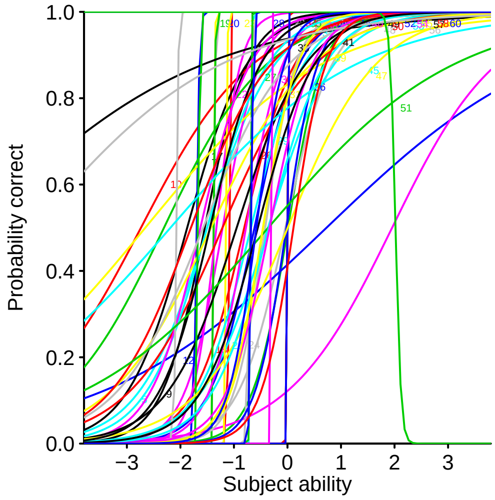


Figure 7.7: Probability that a worker having a given ability (x-axis) will correctly answer a given question (numbered colored lines); fitted using item response theory. Data from Chapman et al.[324] Github–Local

Some coding constructs can be incrementally made more difficult to answer, or support alternative representations.

A study by Ajami, Woodbridge and Feitelson[23] investigated the performance of 222 professional developers when answering questions about the behavior of code snippets. The questions contained 28 `if-statement` snippets (out of 41), whose conditions tested against a disjoint range of values, expressed either as a single expression, or as a sequence of nested `if-statements`. For instance:

```
// linear form:
       if (x>0 && x<10 || x>20 && x<30 || x>40 && x<50) {
          print("1");
       } else {
          print("2");
       }
// equivalent nested form:
       if (x>0) {
          if (x<10) {
             print ("1");
          } else {
             print ("2");
          }
       } else if (x>20)
          ...
```

The test performed by each snippet involved between two and four discrete ranges (in one condition, or via nested `if-statements`), and some tests involved negated subexpressions. Modeling subject response time finds that this increases as the number of ranges checked increases (by around 15%, or 3 seconds in an additive model), and decreases for nested `if-statements` (by around 10%, or 2 seconds in an additive model). Modeling answer correctness finds that this decreases as the number of ranges checked increases; see Github–sourcecode/Complexity18EmpSE.R.

The result from this experiment (if replicated) provides one set of inputs into the analysis of the factors involved in the use of `if-statements`.

Linear reasoning is discussed in section 2.6.2.

A condition testing for inclusion within a continuous single range can be written in many ways, including the following:

```
if (x > u && x < e) ...
if (u < x && x < e) ...
if (x < e && u > x) ...
```

A study by Jones[918] investigated developer performance when answering questions about the relative value of two variables, given information about their values relative to a third

[vi] For unknown reasons, the response to problem 51 is the opposite of that expected.

variable. A total of 844 answers were given by 40 professional developers, with 40 incorrect answers (no timing information). With so few incorrect answers it is not possible to distinguish performance differences due to the form of the condition.

7.1.3 Exponential or Power law

Many source code measurements can be well fitted by an exponential and/or power law equation. Ideally, the choice of equation is driven by the theory describing the processes that generated the measurements, but such a theory may not be available. Sometimes, the equation chosen may be based on the range of values considered to be important.

Figure 7.8 shows the same data fitted by an exponential (upper) and power law (lower); note different x-axis scales are used. The choice of equation to fit might be driven by the range of nesting depths considered to be important (or perhaps the range considered to be unimportant), and the desire to use a simple, brand-name, equation, i.e., the complete range of data may be fitted by a more complicated, and less well-known, equation.

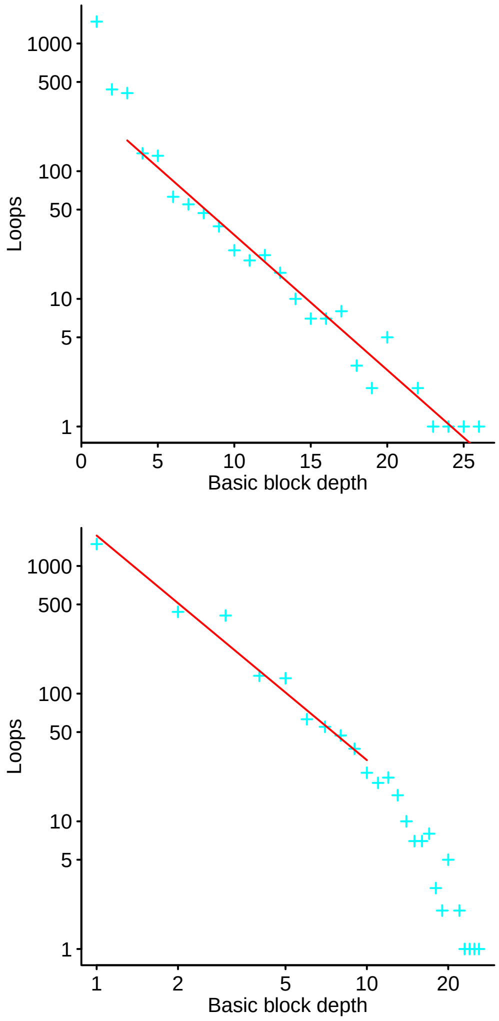


Figure 7.8: Number of for-loops, in C source, whose enclosed compound-statement contained basic blocks nested to a given depth; with fitted exponential (upper) and power law (lower). Data kindly provided by Pani.[1424] Github–Local

In figure 7.37 an exponential was fitted, based on the assumption that the more deeply nested constructs were automatically generated, and that the probability of encountering a `selection-statement` in human written code did not vary with nesting depth.

Section 11.5.1 discusses the analysis needed to check whether it is statistically reasonable to claim that data is fitted by a power law.

7.2 Desirable characteristics

What characteristics are desirable in source code?

This section assumes that desirable characteristics are those that minimise one or more developer resources (e.g., cost, time), and developer mistakes (figure 6.9 suggests that most of the code containing mistakes is modified/deleted before a fault is reported). Desirable characteristics include:

- developers' primary interaction with source code is reading it to obtain the information needed to get a job done. The resources consumed by this interaction can be reduced by:
  - reducing the amount of code that needs to be read, or the need to remember previously read code,
  - increasing the memorability of the behavior of code reduces the cost of rereading it to obtain a good enough understanding,
  - reducing the cognitive effort needed to extract a good enough understanding of the behavior of what has been read,
  - being consistent to support the use of existing information foraging[1471] skills, and reducing the need to remember exceptions,
- use of constructs whose behavior is the same across implementations (which has to be weighed against the benefits provided by use of implementation specific constructs):
  - reduces familiarisation costs for developers new to a project,
  - recruitment is not restricted to developers with experience of a specific implementation; see section 3.3.2,
- interacts with people's propensity to make mistakes in a fail-safe way:
  - robust in the sense that any mistake made is less likely to result in a fault experience,
  - use of constructs that contain redundant information, which provides a mechanism for consistency checking,
  - use of constructs that are fragile and likely to noticeably fail unless used correctly,
- executes within acceptable resource limits. While performance bottlenecks may arise from interaction between the characteristics of the input and algorithm choices, there are domains where individual coding constructs can have a noticeable performance impact, e.g., SQL queries,[243] or might be believed to have such an impact (see fig 1.14 and fig 11.22).

Use of the terms *maintainability*, *readability* and *testability* are often moulded around the research idea, or functionality, being promoted by an individual researcher, i.e., they are essentially a form of marketing.

The production of books, reports, memos and company standards containing suggestions and/or recommendations for organizing source code, and the language constructs to avoid (or use), so-called *coding guidelines*, is something of a cottage industry, e.g., for C.[421, 621, 632, 777, 848, 898, 977, 1020, 1216, 1274, 1476, 1477, 1524, 1528, 1536, 1644, 1726, 1731, 1767]

Stylistically, guideline documents are often more akin to literary criticism than engineering principles, i.e., they express personal opinions that are not derived from evidence (other than perhaps episodes in a person's life). Recommendations against the use of particular language constructs are sometimes based on the construct repeatedly appearing in fault reports; however, the possibility that the use of alternative constructs will produce more/fewer reported faults is rarely included in the analysis, i.e., the current usage may be the least bad of the available options. The C preprocessor is an example of frequently criticised functionality,[1239] that is widely used because of the useful functionality it uniquely provides.

While several ways of implementing the required functionality may be possible, at the time of writing there is little if any evidence available showing that any construct is more/less likely to have some desirable characteristic, compared to another, e.g., less costly to modify or to understand by future readers of the code, or less likely to be the cause of mistakes.

7.2.1 The need to know

How might source code be organized to minimise the expenditure of cognitive effort per amount of code produced?[vii]

One technique for reducing the expenditure of cognitive effort is to reduce the amount of code that a developer needs to process to get a job done, e.g., reading code to understanding it and writing new code.

How might a developer know whether it is necessary to understand code without first having some understanding of it?

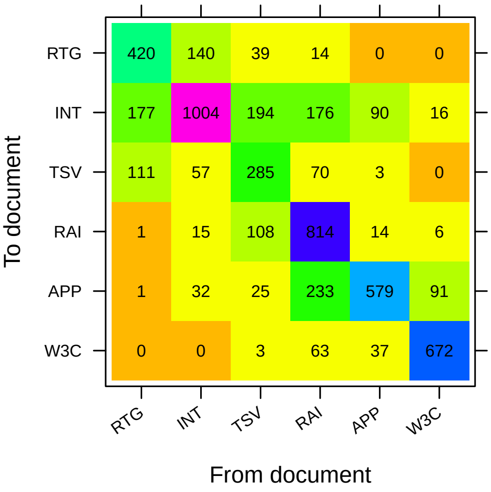
Many languages support functionality for breaking source up into self-contained units/modules,[viii] each having a defined interface; these self-contained units might be functions/methods, classes, files, etc. Language vary in the functionality they provide to allow developers to control the visibility of identifiers defined within a module, e.g., `private` in Java.

The concept of *information hiding* is sometimes used in connection with creating interfaces and associated implementations. This term misrepresents the primary item of interest, which is what developers need to know, not how much information is hidden.

Modularization is a technique used in other domains that build systems from subcomponents, including:

- hardware systems use modularization to make it easier/cheaper to replace broken or worn out components, and to simplify the manufacturing process (as well as manufacturing costs). These issues are not applicable to software, which does not wear out, and has essentially zero manufacturing costs, however, the interface to the world in which the software operates may change in a way that creates a need to modify the software,

- biological systems where connections between components have a cost (e.g., they cause delays in a signalling pathway), and modularity is an organizational method that reduces the number of connections needed;[366] modularity as a characteristic that makes it easier to adapt more quickly when the environment changes may be an additional benefit, rather than the primary driver towards modularity. Simulations[1247] have found that, for non-trivial systems, a hierarchical organization reduces the number of connections needed (for a viable implementation).



	RTG	INT	TSV	RAI	APP	W3C
RTG	420	140	39	14	0	0
INT	177	1004	194	176	90	16
TSV	111	57	285	70	3	0
RAI	1	15	108	814	14	6
APP	1	32	25	233	579	91
W3C	0	0	3	63	37	672

Figure 7.9: Number of citations from Standard documents within protocol level, to documents in the same and other levels (RTG routing, INT internet, TSV transport, RAI realtime applications and infrastructure, APP Applications, W3C recommendations). Data from Simcoe.[1686] Github–Local

[vii] There is too much uncertainty around measuring quantity of functionality provided to make this a viable end-point.

[viii] The term *module* is given a specific meaning in some languages.

Minimizing the need to know is one component of the larger aim of minimising the expenditure of cognitive effort per amount of code produced, which is one component of the larger aim of maximizing overall ROI, i.e., an increase in the need to know is a cost that may be a worthwhile trade-off for a greater benefit elsewhere.

A study by Simcoe[1686] investigated the modularity of communication protocol standards involved in the implementation of the Internet. Figure 7.9 shows the number of citations from IEFT and W3C Standard documents, grouped by protocol layer, that reference Standard documents in the same and other layers. Treating citations as a proxy for dependencies: 89% are along the main diagonal (a uniform distribution would produce 17%); dependencies discovered during implementation may substantially change this picture.

Dependencies between units of code can be used to uncover possible clusters of related functionality. One dependency is function/method calls between larger units of code, with units of code making many of the same calls likely to have something in common.

A study by Almossawi[41] investigated the architectural complexity of early versions of Firefox. The source code of the gfx module in Firefox version 20 is contained in 2,664 files, and makes 14,195 calls from/to methods in these files. Figure 7.10 shows one clustering of files based on the number of from/to method calls.

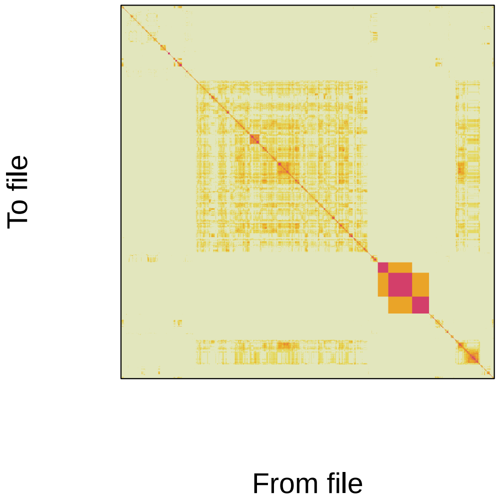


Figure 7.10: A clustering of the 2,664 files containing from/to method calls in the gfx module of Firefox version 20. Data kindly provided by Almossawi.[41] Github–Local

7.2.2 Narrative structures

People are inveterate storytellers, and narrative is another way of interpreting source code. People tell stories about their exploits, and a culture's folktales are passed on to each new generation. The narrative structures present in the folktales told within cultures have many features in common.[1512]

The Aarne-Thompson-Uther Index (ATU) is a classification of 2,399 distinct folktale templates (based on themes, plots and characters). A study by Tehrani[1799] investigated the phylogeny of the European folktale "Little Red Riding Hood" (ATU 333), a very similar folktale from Japan, China and Korea known as "The Tiger Grandmother" which some classify as ATU 123 (rather than ATU 333), and other similar folktales not in the ATU index. Figure 7.11 shows a phylogenetic tree of 58 folktales, based on 72 story characteristics, with 18 classified as ATU 333 (red), 20 as ATU 123 (blue), and 20 not classified (green).

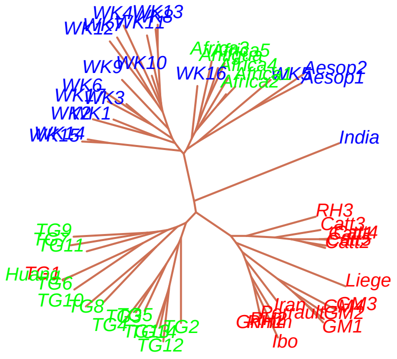


Figure 7.11: Phylogenetic tree of 58 folktales, based on 72 story characteristics; 18 classified as ATU 333 (red), 20 as ATU 123 (blue), and 20 unclassified (green). Data from Tehrani.[1799] Github–Local

The ability of some narrative structures to survive through many retellings, and to spread (or be locally created), suggests they have characteristics that would be desirable in source code, e.g., memorability and immunity to noise (such as introduction of unrelated subplots).

A program's narrative structure (i.e., its functionality) emerges from the execution of selective sequences of code, which may be scattered throughout the source files used to build a program. The source code of programs implemented using an Interactive Fiction approach is essentially the program narrative.[1198] The term *programming plans* is used in some studies.[1948]

A study by Wong, Gokhale and Horgan[1956] investigated the execution of basic blocks, within the 30 source files making up the SHARPE program, when fed inputs that exercised six of the supported features (in turn). Figure 7.12 shows the fraction of basic blocks in each file executed when processing input exercising a particular feature.

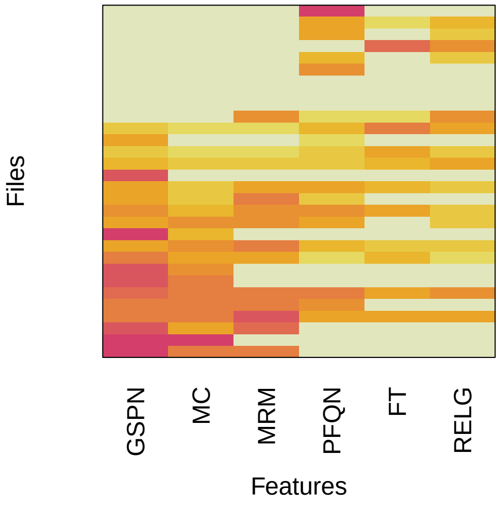


Figure 7.12: Heat map of the fraction of each of 30 files' basic blocks executed when performing a given feature of the SHARPE program. Data from Wong et al.[1956] Github–Local

The narratives of daily human activity are constrained by the cost of moving in space, and the feasibility of separating related activities in time. The same constraints apply to mechanical systems, along with the ability to be manufactured at a profit.

The narratives achievable in a software system are constrained by the cognitive capacity, and knowledge of the people who implemented it, along with the ability to use the system within the available storage and processing capacity.

Languages support a variety of constructs for creating narrative components that can be fitted together, e.g., functions/methods, classes, and generics/templates. The purpose of generics/templates is to provide a means of specifying a general behavior that can later be instantiated for specific cases, e.g., the generic behavior is to return the maximum of a list of values, and a specific instance involves the values having an integer type.

A study by Chen, Wu, Ma, Zhou, Xu and Leung[335] investigated the use of C++ templates, such as the definition and use of new templates by developers, and the use of templates defined in the Standard Template Library. For the five largest projects: around 25% of developer-defined function templates and 15% of class templates were instantiated once, and there were seventeen times as many instantiations of function templates defined in the STL compared to developer-defined function templates (149,591 vs. 8,887). Figure 7.13 shows that a few developer-defined function templates account for most of the template instantiations in a project.

Reasons for the greater use of STL templates include: the library supports the commonly required functionality, and documentation on the available templates is readily available. Developer-defined templates are likely to be application specific, and documentation on them may not be readily available to other developers. The study found that most templates were defined by a few project developers, which may be an issue of developer education, or applications only needing specific templates in specific cases.

Studies of the introduction of generics in C♯[985] and Java[1427] found that while developers made use of functionality defined in libraries using generics, they rarely defined their own generic classes. Also, existing code in most projects was not refactored to use generics, and the savings from refactoring (measured in lines of code) was small.

How much source code appears in the implementation of distinct components of a narrative?

A study by Landman, Serebrenik, Bouwers and Vinju[1065] measured 19K open source Java projects, and the 9.6K packages (written in C) contained in a Linux distribution. Figure 7.14 shows the number Java methods and C functions containing a given number of source lines. While a power law provides a good fit, over some range, of both sets of data, the majority of C functions have a size distribution that differs from Java; see Github–sourcecode/Landman_m_ccsloc.R.

Figure 7.14 shows that most Java methods are very short, while the size range of the majority C functions is much wider (i.e., four to ten lines); figure 7.23 shows that 50% of Java source occurs within methods containing four lines or less, while in C 50% of source appears in functions containing 114 lines, or less.

One study[316] of function calls in eight C programs found, statically, that function calls in some programs were mostly to functions defined in other files, while calls in the other programs were to functions defined in the same file; the same static intra/inter file call predominance tended to occur at runtime.

Narratives are created and motivated by pressures such as:

- startups seeking to bring a saleable narrative to market as quickly as possible (i.e., a minimum viable product), to enable them to use customer feedback to enrich and extend the narrative,
- companies with established systems seeking to evolve the software to keep it consistent with the real-world narrative within which it has become intertwined,
- open source developers creating narratives for personal enjoyment.

Language tokens (such as identifiers, keywords and integer literals) are not the source code equivalent of words, but more like the phonemes (a distinct unit of sound) that are used to form a word. Most lines only contain a few tokens (see fig 8.4), and might form a distinct unit of thought or act as a connection between the adjacent lines.

7.2.3 Explaining code

Before a developer can successfully complete a source code related task, they have to invest in obtaining a good enough explanation of the behavior of the appropriate code. The nature of the task is likely to drive the approach the developer takes, to the explanation process[847] (the term *understanding* is often used by software developers, and *comprehension* is used in prose related research).

Human reasoning is discussed in section 2.6.

A small modification may only require understanding how the code implements the functionality it provides (e.g., algorithms used), while a larger change may require searching for existing code that could be impacted by the change; fixing a reported fault is a search process that often involves localised understanding.

Figure 7.15 shows the size of commits involved in fixing reported faults in Linux; also see fig 8.15.

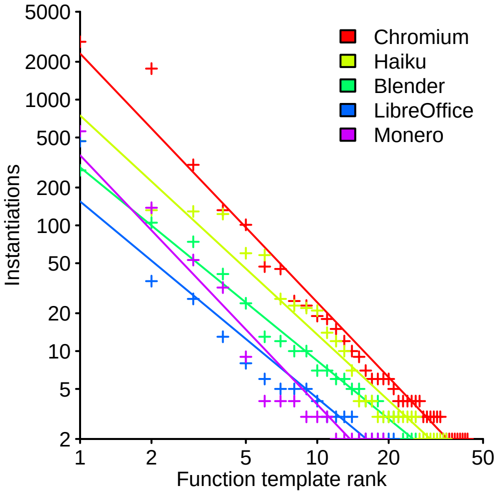


Figure 7.13: Sorted number of instantiations of each developer-defined C++ function template; fitted regression lines have the form: $\mathit{Instantiations} \propto \mathit{template\_rank}^{-K}$, where K is between 1.5 and 2. Data from Chen et al.[335] Github–Local

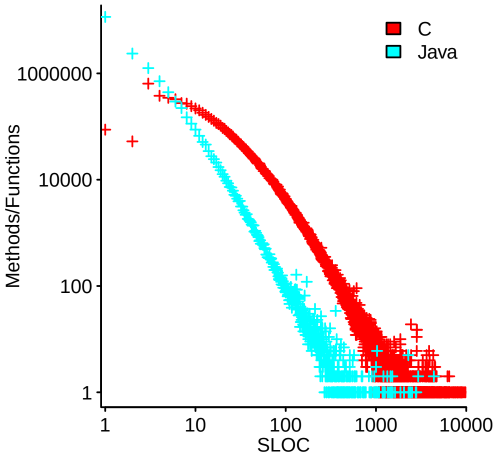


Figure 7.14: Number of methods/functions containing a given number of source lines; 17.6M methods, 6.3M functions. Data kindly provided by Landman.[1065] Github–Local

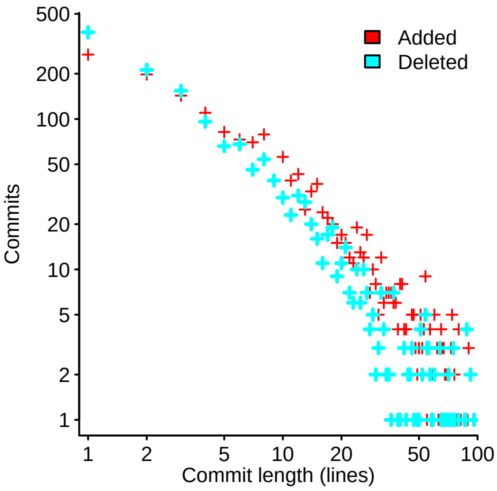


Figure 7.15: Number of commits of a given length, in lines added/deleted to fix various faults in Linux file systems. Data from Lu et al.[1152] Github–Local

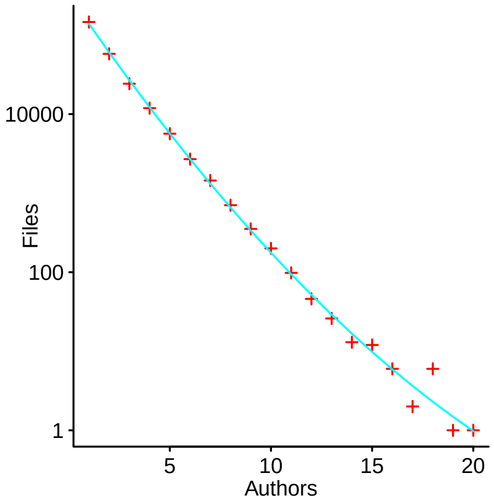


Figure 7.16: Number of files, in Eclipse projects, that have been modified by a given number of people; line is a fitted regression model of the form: $Files \propto e^{-0.87authors+0.01authors^2}$. Data from Taylor.[1797] Github–Local

The procedure is really quite simple. First you arrange things into different groups depending on their makeup. Of course, one pile may be sufficient depending on how much there is to do. If you have to go somewhere else due to lack of facilities that is the next step, otherwise you are pretty well set. It is important not to overdo any particular endeavor. That is, it is better to do too few things at once than too many. In the short run this may not seem important, but complications from doing too many can easily arise. A mistake can be expensive as well. The manipulation of the appropriate mechanisms should be self-explanatory, and we need not dwell on it here. At first the whole procedure will seem complicated. Soon, however, it will become just another facet of life. It is difficult to foresee any end to this task in the immediate future, but then one never can tell.

For some tasks the cost-effectiveness of understanding the behavior of a complete program will decrease as the size of the program increases (because the task can be completed with an understanding of a subset of the program's behavior). one study[1142] based around a 250-line Fortran program found some developers used an as-needed strategy, and others attempted to understand the complete program.

To what extent do the contents of existing files affect the future coding habits of developers (because they read and learn from the source code contained in one or more of these files)?

A lower bound on the number of times a file has been written is provided by version control check-in history. It is not known, at the time of writing, whether check-outs can be used as a good-enough proxy for the number of times a file is read.

A study by Taylor[1797] investigated author contribution patterns to Eclipse projects. Figure 7.16 shows the number of files modified by a given number of people.

Developers do not understand programs, as such, they acquire beliefs about program behavior; a continuous process involving the creation of new beliefs and the modification of existing ones, with no well-defined ending. The beliefs acquired are influenced by existing beliefs about the programming language it is written in, general computing algorithms, and the application domain.[1653]

People search for meaning and explanations.[970] Developers may infer an *intended meaning* of source code, i.e., a belief about what the meaning that the original author of the code intended to implement. Code understanding is an exercise in obtaining an intended meaning that is assumed to have existed.

Activities that appear to be very complicated, can have a simple, but difficult to discover, explanation. For instance, some hunting rituals intended to select the best hunting location are actually randomization algorithms,[1655] whose effect is to reduce the likelihood of the community over-hunting any location.

What can be done to reduce the cognitive effort that needs to be invested to obtain a good-enough interpretation of the behavior of code?

Source code is an implementation of application requirements. An understanding of the kinds of activities involved within the application domain provides a framework for guiding an interpretation of the intended behavior of a program's source.

A study by Bransford and Johnson[241] investigated the impact of having a top-level description on the amount of information subjects' remembered about a task. Try to give a meaning to the task described in the outer margin, while remembering what is involved (taken from the study).

Table 7.3 shows that subjects' recalled over twice as much information, if they were given a meaningful phrase (the topic), before reading the passage. The topic of the passage in the margin is ʍɐsɥıuƃ cloʇɥǝs.

	No Topic	Topic Given	Topic Given	Maximum
	Given	After	Before	Score
Comprehension	2.29 (0.22)	2.12 (0.26)	4.50 (0.49)	7
Recall	2.82 (0.60)	2.65 (0.53)	5.83 (0.49)	18

Table 7.3: Mean comprehension rating and mean number of ideas recalled from passage (standard deviation in parentheses). Adapted from Bransford and Johnson.[241]

In one study[1446] investigating subject performance, answering questions about the code contained in a 200-line program they had studied; developers who had built an application domain model from the code performed best.

Some form of understanding may be achieved by assembling basic units of information into a higher level representation. In human languages, native speakers effortlessly operate on words, which are a basic unit of understanding. The complexity of human languages, which have to be processed in real-time while listening to the speaker, is constrained by the working memory capacity of those involved in the communication activity.[666,785] The capacity limits that make it difficult for speakers to construct complicated sentences, in real-time, are a benefit for listeners (who share similar capacity limits).

A study by Futrell, Mahowald and Gibson[633] investigated dependency length (the distance between words, in a sentence, that depend on each other; see figure 7.17) in the use of

37 human languages. The results suggest that speakers attempt to minimise dependency length.

Sentence complexity[341] has a variety of effects on human performance. A study by Kintsch and Keenan[994] asked subjects to read single sentences, each containing the same number of words, but varying in the number of propositions they contained (see figure 7.18). The time taken to read each sentence and recall it (immediately after reading it), was measured.

Figure 7.19 shows reading time (in seconds) for sentences containing a given numbers of propositions (blue), and reading time for when a given number of propositions were recalled by subjects (red); with fitted regression models. A later study[995] found that reader performance was also affected by the number of word concepts in the sentence, and the grammatical form of the propositions (subordinate or superordinate).

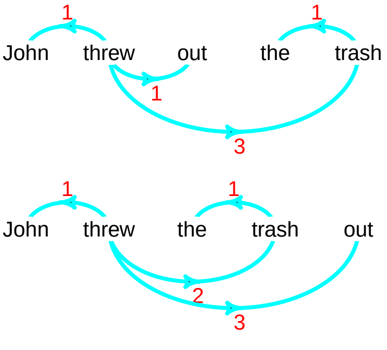
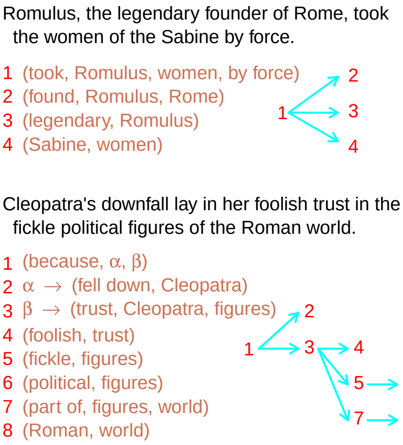


Figure 7.17: Two sentences, with their dependency representations; upper sentence has total dependency length six, while in the lower sentence it is seven. Based on Futrell et al.[633] Github–Local


Romulus, the legendary founder of Rome, took the women of the Sabine by force.

1 (took, Romulus, women, by force)
2 (found, Romulus, Rome)
3 (legendary, Romulus)
4 (Sabine, women)

Cleopatra's downfall lay in her foolish trust in the fickle political figures of the Roman world.

1 (because, α, β)
2 α → (fell down, Cleopatra)
3 β → (trust, Cleopatra, figures)
4 (foolish, trust)
5 (fickle, figures)
6 (political, figures)
7 (part of, figures, world)
8 (Roman, world)

Figure 7.18: One sentence containing four, and the other eight propositions, along with their propositional analyses. Based on Kintsch et al.[994] Github–Local

7.2.4 Memory for material read

Increasing the likelihood that information extracted from code will be accurately recalled later reduces the costs associated with having to reextract the information.

What do people remember about the material they have read (human memory systems are discussed in section 2.4)?

Again, the only detailed experimental data available is from studies based on human language prose.

Studies[240] have found that in the short term, syntax is remembered (i.e., words), while over the longer term mostly semantics is remembered (i.e., the meaning); explicit verbatim memory for text does occur.[752] Readers might like to try the following test (based on Jenkins[905]); part 1: A line at a time, 1) read the sentence on the left, 2) look away and count to five, 3) answer the question on the right, and 4) repeat process for the next line.

The girl broke the window on the porch.	Broke what?
The hill was steep.	What was?
The cat, running from the barking dog, jumped on the table.	From what?
The tree was tall.	Was what?
The old car climbed the hill.	What did?
The cat running from the dog jumped on the table.	Where?
The girl who lives next door broke the window on the porch.	Lives where?
The car pulled the trailer.	Did what?
The scared cat was running from the barking dog.	What was?
The girl lives next door.	Who does?
The tree shaded the man who was smoking his pipe.	What did?
The scared cat jumped on the table.	What did?
The girl who lives next door broke the large window.	Broke what?
The man was smoking his pipe.	Who was?
The old car climbed the steep hill.	The what?
The large window was on the porch.	Where?
The tall tree was in the front yard.	What was?
The car pulling the trailer climbed the steep hill.	Did what?
The cat jumped on the table.	Where?
The tall tree in the front yard shaded the man.	Did what?
The car pulling the trailer climbed the hill.	Which car?
The dog was barking.	Was what?
The window was large.	What was?

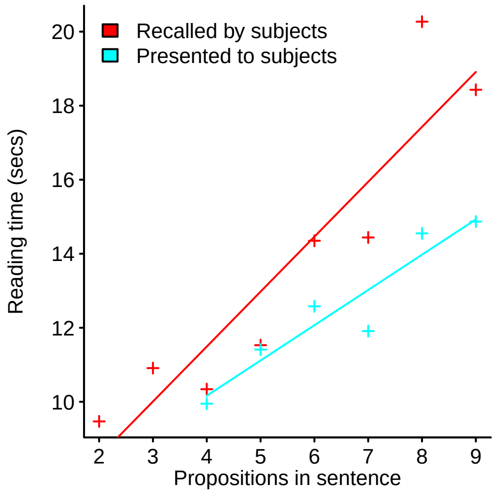


Figure 7.19: Mean reading time (in seconds) for sentences containing a given number of propositions, and as a function of the number of propositions recalled by subjects; with fitted regression models. Data extracted from Kintsch et al.[994] Github–Local

You have now completed part 1. Please do something else for a minute, or so, before moving on to part 2 (which follows immediately below).

Part 2: when performing this part, do not look at the sentences above, from part 1; look at the sentences below. Now, a line at a time, 1) read the sentence on the left, 2) if you think that sentence appeared as a sentence in part 1 express your confidence level by writing a number between one and five (with one expressing very little confidence, and five expressing a lot of confidence in the decision) next to **old**, otherwise write a number representing your confidence level next to **new**, and 3) repeat process for the next line.

```
The car climbed the hill.                                old___, new ___
The girl who lives next door broke the window.           old___, new ___
The old man who was smoking his pipe climbed the steep hill.  old___, new ___
The tree was in the front yard.                          old___, new ___
The window was on the porch.                                         old___, new ___
The barking dog jumped on the old car in the front yard.             old___, new ___
The cat was running from the dog.                                    old___, new ___
The old car pulled the trailer.                                      old___, new ___
The tall tree in the front yard shaded the old car.                  old___, new ___
The scared cat was running from the dog.                             old___, new ___
The old car, pulling the trailer, climbed the hill.                  old___, new ___
The girl who lives next door broke the large window on the porch. old___, new ___
The tall tree shaded the man.                                        old___, new ___
The cat was running from the barking dog.                            old___, new ___
The cat was old.                                                     old___, new ___
The girl broke the large window.                                     old___, new ___
The car climbed the steep hill.                                      old___, new ___
The man who lives next door broke the window.                        old___, new ___
The cat was scared.                                                  old___, new ___
```

You have now completed part 2. Count the number of sentences you judged to be **old**.

The surprise is that all the sentences are new.

What is thought to happen is that while reading, people abstract and remember the general ideas contained in sentences. In this case, they are based on the four *idea sets*: 1) “The scared cat running from the barking dog jumped on the table.”, 2) “The old car pulling the trailer climbed the steep hill.”, 3) “The tall tree in the front yard shaded the man who was smoking his pipe.”, and 4) “The girl who lives next door broke the large window on the porch.”.

A study by Bransford and Franks[240] investigated subjects’ confidence of having previously seen a sentence. Sentences contained either one idea unit (e.g., “The cat was scared.”), two idea units (e.g., “The scared cat jumped on the table.”), three idea units (e.g., “The scared cat was running from the dog.”), or four idea units (e.g., “The scared cat running from the barking dog jumped on the table.”). Subjects saw 24 sentences, after a 4-5 minute break they were shown 28 sentences (24 of which were new sentences), and asked to rank their confidence of having previously seen the sentence (on a 1 to 5 scale).

Figure 7.20 shows that subjects’ confidence of having previously seen a sentence increases with the number of idea units it contains. New sentences contained one or more idea units contained in previously seen sentences. The results are consistent with subject confidence level being driven by the number of previously seen idea units in a sentence, rather than the presence of new idea units.

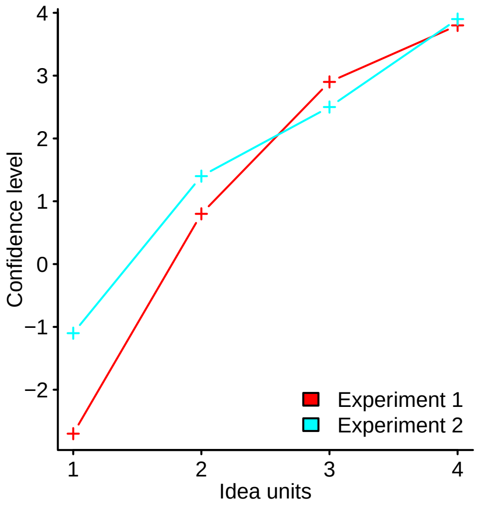


Figure 7.20: Subject confidence level, on a one to five scale (yes positive, no negative), of having previously seen a sentence containing a given number of idea units (experiment 2 was a replication of experiment 1, plus extra sentences). Data extracted from Bransford et al.[240] Github–Local

People use their experience with the form and structure of often repeated sequences of actions, to organize the longer-term memories they form about them. The following studies illustrate the effect that a person’s knowledge of the world can have on their memory for what they have read, particularly with the passage of time, and their performance in interpreting sequences of related facts they are presented with:

- A study by Bower, Black and Turner[229] gave subjects a number of short stories describing various activities to read, such as visiting the dentist, attending a class lecture, going to a birthday party, (i.e., scripts). Each story contained about 20 actions, such as looking at a dental poster, having teeth X-rayed, etc. After a 20-minutes interval, subjects were asked to recall actions contained in the stories.

  The results found that around a quarter of recalled actions might be part of the script, but were not included in the written story. Approximately 7% of recalled actions were not in the story, and would not be thought to belong to the script.

  A second experiment involved subjects reading a list of actions, which in the real world, would either be expected to occur in a known order or not be expected to have any order (e.g., the order of the float displays in a parade). The results showed that, within ordered scripts, actions that occurred at their expected location were recalled 50% of the time, while actions occurring at unexpected locations were recalled 18% of the time at that location. The recall rate for unordered scripts (i.e., the controls) was 30%.

- A study by Graesser, Woll, Kowalski and Smith[717] read subjects stories representing scripted activities (e.g., eating at a restaurant). The stories contained actions that varied in the degree to which they were typical of the script (e.g., Jack sat down at the table, Jack confirmed his reservation, and Jack put a pen in his pocket).

  Table 7.4 shows the results; recall was not affected by typicality over short periods of time, but after one week recall of atypical actions dropped significantly. Recognition

Memory Test	**Typical (30 mins)**	**Atypical (30 mins)**	**Typical (1 week)**	**Atypical (1 week)**
Recall (correct)	0.34	0.32	0.21	0.04
Recall (incorrect)	0.17	0.00	0.15	0.00
Recognition (correct)	0.79	0.79	0.80	0.60
Recognition (incorrect)	0.59	0.11	0.69	0.26

Table 7.4: Probability of subjects recalling or recognizing, typical or atypical actions present in stories read to them, at two time intervals (30 minutes and 1 week) after hearing them. Based on Graesser et al.[717]

performance (i.e., subjects were asked if a particular action occurred in the story) for typical vs. atypical actions was less affected by the passage of time.

- A study by Dooling and Christiaansen[499] asked subjects to read a short biography containing 10 sentences. The only difference between the biographies was that in some cases the name of the character was fictitious (i.e., a made up name), while in other cases it was the name of an applicable famous person. For instance, one biography described a ruthless dictator, and used either the name Gerald Martin or Adolph Hitler.

  After 2-days, and then after 1-week, subjects were given a list of 14 sentences (seven sentences that were included in the biography they had previously read, and seven that were not included), and asked to specify, which sentences they had previously read.

  To measure the impact of subjects' knowledge about the famous person, on recognition performance, some subjects were given additional information. In both cases the additional information was given to the subjects who had read the biography containing the made up name (e.g., Gerald Martin). The *before* subjects were told just before reading the biography that it was actually a description of a famous person and given that persons name (e.g., Adolph Hitler). The *after* subjects were told just before performing the recognition test that the biography was actually a description of a famous person and given that persons name (they were given one minute to think about what they had been told).

  Figure 7.21 shows that the results are consistent with the idea that remembering is constructive. After a week subjects memory for specific information in the passage was lost, and under these conditions recognition of sentences is guided by subjects' general knowledge. Variations in the time between reading the biography, and identity of a famous character being revealed, affected the extent to which subjects integrated this information.

These results suggest that it is a desirable characteristic (i.e., more information, more accurately recalled), for the contents of scripted stories to be consistent with readers' prior knowledge and expectations (e.g., events that occur and their relative ordering). The extent to which source code can be organised in this way will depend on the application requirements and any demands for algorithmic efficiency.

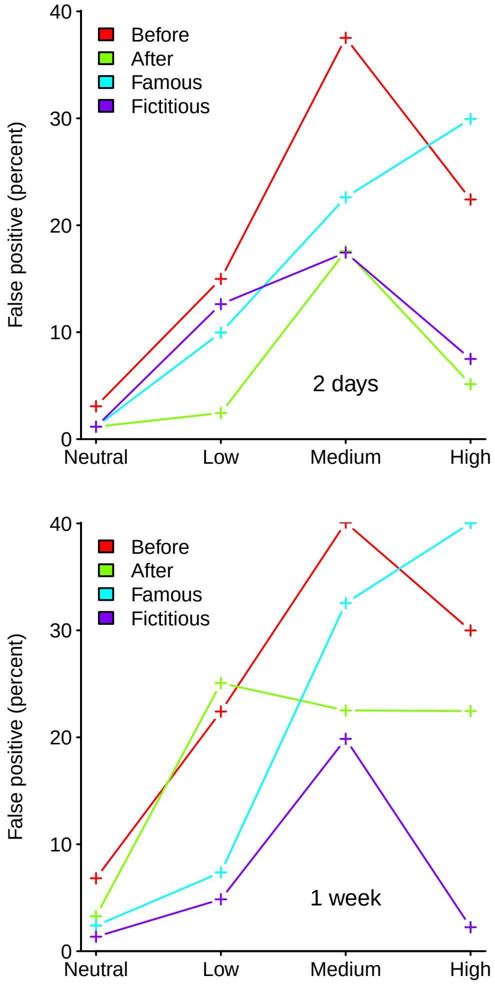


Figure 7.21: Percentage of false-positive recognition errors for biographies having varying degrees of thematic relatedness to the famous person, in *before*, *after*, *famous*, and *fictitious* groups. Data extracted from Dooling et al.[499] Github–Local

7.2.5 Integrating information

Units of source code (e.g., statements) are sequenced in ways that result in a behavioral narrative emerging during program execution. To modify an existing narrative a developer needs to acquire a good enough understanding of how the units of code are sequenced to produce the emergent behavior. Information has to be extracted and integrated into a mental model of program operation.

Which factors have the largest impact on the cognitive effort needed to integrate source code information into a mental model? Studies[993, 1230] of prose comprehension provide some clues.

The process of integrating two related items of information involves acquiring the first item, and keeping it available for recall while processing other information, until the second item is encountered; it is then possible to notice that the two items are related, and act on this observation.

A study by Daneman and Carpenter[425] investigated the connection between various measures of subjects' working memory span, and their performance on a reading comprehension task. The two measures of working memory used were the *word span* and *reading span*. The word span test is purely a measure of memory usage. In the reading span test subjects have to read, out loud, sequences of sentences while remembering the last word of each sentence, which have to be recalled at the end of the sequence. In the test, the number of sentences in each sequence is increased until subjects are unable to successfully recall all the last words.

The reading comprehension test involves subjects reading a narrative passage containing approximately 140 words, and then answering questions about facts and events described in the passage. Passages are constructed such that the distance between the information needed to answer questions varies. For instance, the final sentence of the passage might contain a pronoun (e.g., she, her, he, him, or it) referring to a noun appearing in a previous sentence, with different passages containing the referenced noun in either the second, third, fourth, fifth, sixth, or seventh sentence before the last sentence.

In the excerpt: " . . . river clearing . . . The proceedings were delayed because the leopard had not shown up yet. There was much speculation as to the reason for this midnight alarm. Finally, he arrived, and the meeting could commence." the question: "Who finally arrived?" refers to information contained in the last and third to last sentence; the question: "Where was the meeting held?" requires the recall of a fact.

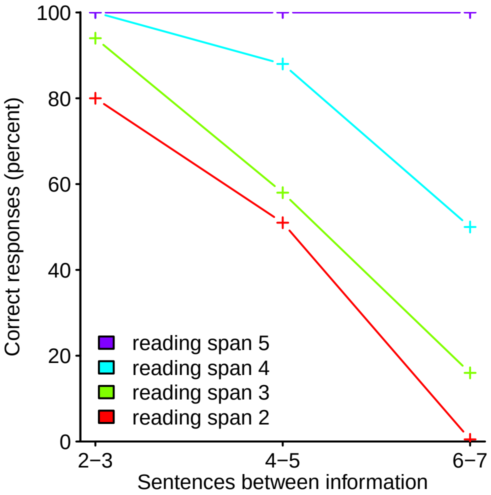


Figure 7.22: Percentage of correct responses in a reading comprehension test, for subjects having a given reading span, using the pronoun reference questions as a function of the number of sentences (x-axis) between the pronoun and the referent noun. Data extracted from Daneman et al.[425] Github–Local

Figure 7.22 show the relationship between subject performance in the reading span test and the reading comprehension test. A similar pattern of results was obtained when the task involved listening, rather than reading. A study by Turner and Engle[1836] found that having subjects verify simple arithmetic identities, rather than a reading comprehension test, did not alter the results. However, altering the difficulty of the background task (e.g., using sentences that required more effort to comprehend) reduced performance.

As a coding example, given the following three assignments, would moving the assignment to x after the assignment to y, reduce the cognitive effort needed to comprehend the value of the expression assigned to z?

```
x = ex_1 + ex_2;           /* Could be moved to after assignment to y. */
y = complicated_expression; /* No dependencies on previous statement.   */
z = y + ex_1;
```

This question assumes that ex_2 does not appear prior to the assignment to x, in which case there may be a greater benefit to this assignment appearing close to the prior usage, rather than close to the assignment to z; at the time of writing there is little if any evidence available that might be used to help answer these questions.

Is reader cognitive effort reduced by having a single complex statement, rather than several simpler statements?

A study by Daneman and Carpenter[426] investigated subjects performance when integrating information within a single sentence, compared to across two sentences, e.g., "There is a sewer near our home who makes terrific suits" (this is what is known as a *garden path* sentence), and "There is a sewer near our home. He makes terrific suits." The results found that a sentence boundary can affect comprehension performance. It was proposed that this performance difference was caused by readers purging any verbatim information they held in working memory, about a sentence, on reaching its end. The availability of previously read words, in the single sentence case, making it easier to change an interpretation, based of what has already been read.

Putting too much information in one sentence has costs. A study by Gibson and Thomas[667] found that subjects were likely to perceive complex ungrammatical sentences as being grammatical. Subjects handled complex sentence that exceeded working memory capacity by forgetting parts of the syntactic structure of the sentence, to create a grammatically correct sentence.

A study by Kintsch, Mandel, and Kozminsky[996] investigated the time taken to read and summarize 1,400 word stories. The order of the paragraphs (not the sentences) in the text seen by some subjects was randomized. The results showed that while it was not possible to distinguish between the summaries produced by subjects reading ordered vs. randomised stories, reading time for randomly ordered paragraphs was significantly longer (9.05 minutes vs. 7.34).

A study by Ehrlich and Johnson-Laird[525] asked subjects to draw diagrams depicting the spatial layout of everyday items specified by a sequence of sentences. The sentences varied in the extent to which an item appearing as the object (or subject, or not at all) in one sentence appeared as the subject (or object, or not at all) in the immediately following sentence. For instance, there is referential continuity in the sentence sequence "The knife is in front of the pot. The pot is on the left of the glass. The glass is behind the dish.", but not in the sequence “The knife is in front of the pot. The glass is behind the dish. The pot is on the left of the glass.”

The results found that when the items in the sentence sequences had referential continuity 57% of the diagrams were correct, compared to 33% when there was no continuity. Most of the errors for the non-continuity sentences were items missing from the diagram drawn, and subjects reported finding it difficult to remember the items as well as the relationship between them.

Most functions contain a few lines (see fig 7.14), and figure 7.23 shows that, depending on language, most of a program’s code appears in the shorter functions.

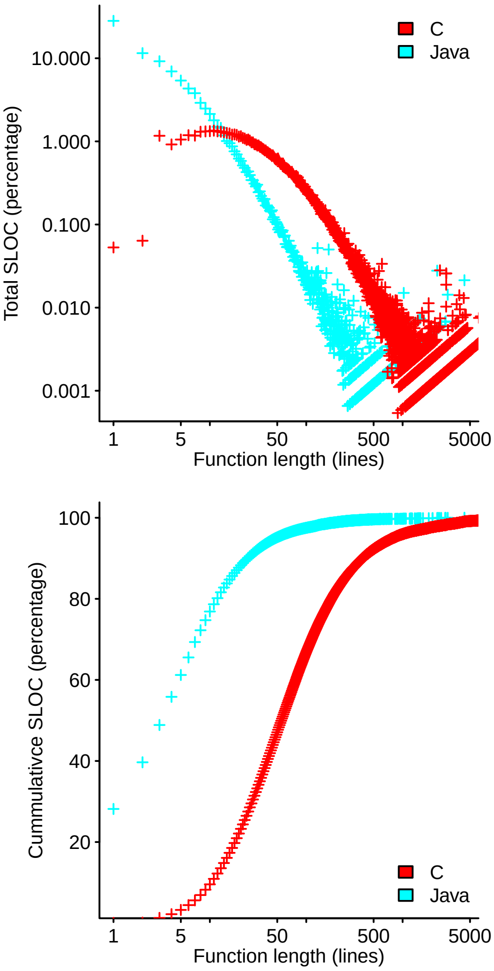


Figure 7.23: Lines of code (as a percentage of all lines of code in the language measured) appearing in C functions and Java methods containing a given number of lines of code (upper); cumulative sum of SLOC percentage (lower). Data kindly provided by Landman.[1065] Github–Local

7.2.6 Visual organization

The human brain contains several regions that perform specific kinds of basic processing of the visual input, along with regions that use this processed information to create higher level models of the visual scene; see section 2.3.

High level visual attention is a limited resource, as illustrated by some of the black circles in figure 7.24 not being visible when not directly viewed. How might the visual layout of source code be organized to reduce the need for conscious attention, by making use of the lower level processing capability that is available (e.g., indenting the start of adjacent lines to take advantage of preattentive detection of lines)?

People’s ability to learn means that, with practice, they can adapt to handle a wide variety of disparate visual organizations of character sequences. The learning process requires practice, which takes time. Using a visual organization likely to be familiar to developers reduces the start-up cost of adapting to a new layout, i.e., prior experience is used to enable developer performance to start closer to their long-term performance.

How quickly might people achieve a reading performance, using a new text layout, that is comparable to that achieved with a familiar layout (based on reading and error rate)?

A study by Kolers and Perkins[1022] investigated the extent to which practice improved reading performance. Subjects were asked to read pages of text written in various ways, and the time taken for subjects to read a page of text having a particular orientation was measured; the text could be one of: normal, reversed, inverted, or mirrored text, as in the following:

- Expectations can also mislead us; the unexpected is always hard to perceive clearly. Sometimes we fail to recognize an object because we. . .
- .ekam ot detcepxe eb thgim natiruP dnalgnE weN a ekatsim fo dnik eht saw tI .eb ot serad eh sa yzal sa si nam yreve taht dias ecno nosremE
- These are but a few of the reasons for believing that a person cannot be conscious of all his mental processes. Many other reasons can be. . .
- Several years ago a professor who teaches psychology at a large university had to ask his assistant, a young man of great intelligence. . .

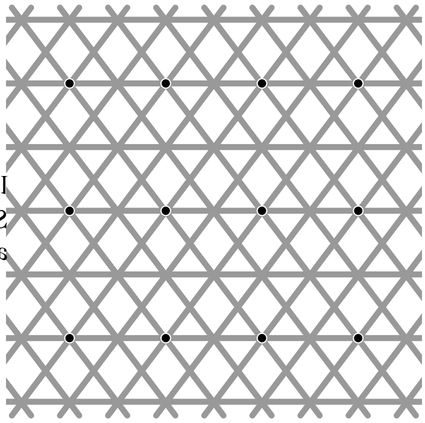


Figure 7.24: Hermann grid, with variation due to Ninio and Stevens[1364] to create an extinction illusion. Github–Local

Figure 7.25 shows the time taken to read a page containing text having a particular orientation. In a study[1021] a year later, Kolers measured the performance of the same subjects, as they read more pages. Performance improved with practice, but this time the subjects had prior experience, and their performance started out better and improved more quickly.

Eye-tracking is used in studies of reading prose to find out where subjects are looking, and for how long; this evidence-based approach is only just starting to be used to study the visual processes involved in processing source code (see fig 2.17), and the impact of factors such as indentation.[144]

7.2.7 Consistency

People subconsciously learn and make use of patterns in events that occur within their environment (see section 2.5). Consistently following patterns of behavior, when writing source code, creates an opportunity for readers to make use of this implicit learning ability. Individual developers write code in a distinct style,[285] even if they cannot articulate every pattern of behavior they follow.

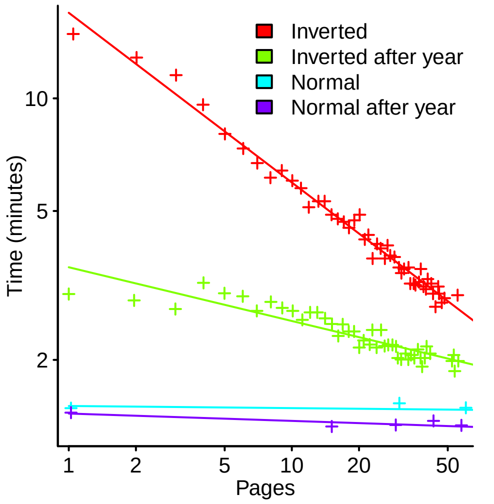


Figure 7.25: Time taken by subjects to read a page of text, printed with a particular orientation, as they read more pages (initial experiment and repeated after one year); with fitted regression lines. Results are for the same six subjects in two tests more than a year apart. Based on Kolers.[1021] Github–Local

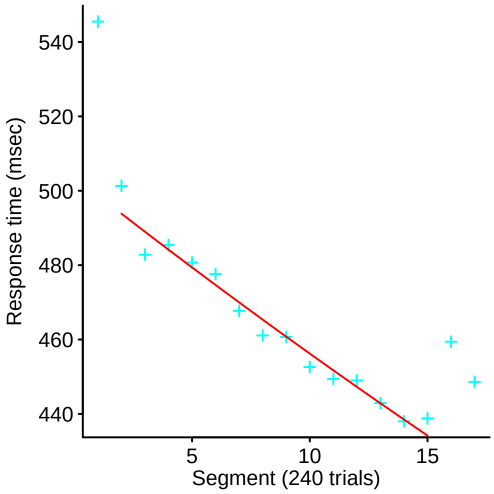


Figure 7.26: Mean response time for each of 17 segments; the regression line fitted to segments 2-15 has the form: $Response\_time \propto e^{-0.1 Segment}$. Data extracted from Lewicki et al.[1108] Github–Local

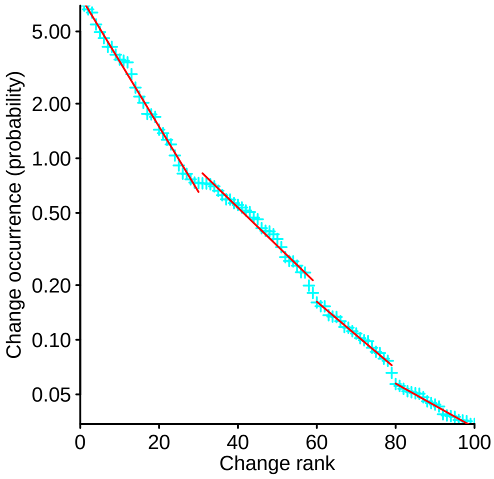


Figure 7.27: Percentage occurrence of kinds of source changes (in rank order), with fitted exponentials over a range of ranks (red lines). Data kindly provided by Martinez.[1200] Github–Local

A study by Lewicki, Hill and Bizot[1108] investigated the impact of implicit learning on subjects' performance, in a task containing no overt learning component. While subjects watched a computer screen, a letter was presented in one of four possible locations; subjects had to press the button corresponding to the location of the letter as quickly as possible. The sequence of locations used followed a consistent, but complex, pattern. The results showed subjects' response times continually improving as they gained experience. The presentation was divided into 17 segments of 240 trials (a total of 4,080 letters). The pattern used to select the sequence of locations was changed after the 15th segment, but subjects were not told about the existence of any patterns of behavior. After completing the presentation subjects were interviewed to find out if they had been aware of any patterns in the presentation; they had not.

Figure 7.26 shows the mean response time for each segment. The consistent improvement, after the first segment, is interrupted by a decrease in performance after the pattern changes on the 15th segment.

A study by Buse and Weimer[278] investigated Computer Science students' opinions of the readability of short snippets of Java source code, rating them on a scale of 1 to 5. The students were taking first, second and third/fourth year Computer Science degree courses or were postgraduates at the researchers' University.

Subjects were not given any instructions on how to rate the snippets for readability, and the attributes that subjects were evaluating when selecting a rating is not known, e.g., were subject ratings based of how readable they personally found the snippets to be, or based on the answer they would expect to give when tested in an exam.

The results show that the agreement between students readability ratings, for short snippets of code, improved as students progressed through course years 1 to 4 of a computer science degree; see Github–developers/readability. The study can be viewed as an investigation of implicit learning, i.e., students learned to rate code against what they had been told were community norms of a quantity called readability.

A study by Corazza, Maggio and Scanniello[393] investigated semantic relatedness, which they called *coherence*, between a method's implementation and any associated comment, i.e., did the method implement the intent expressed in the comment. Five Java programs, containing a total of 5,762 methods, were manually evaluated; the results found that coherence was positively correlated with $\log(comment\_lines)$, and negatively correlated with method *LOC*; see Github–sourcecode/SQJ_2015.R.

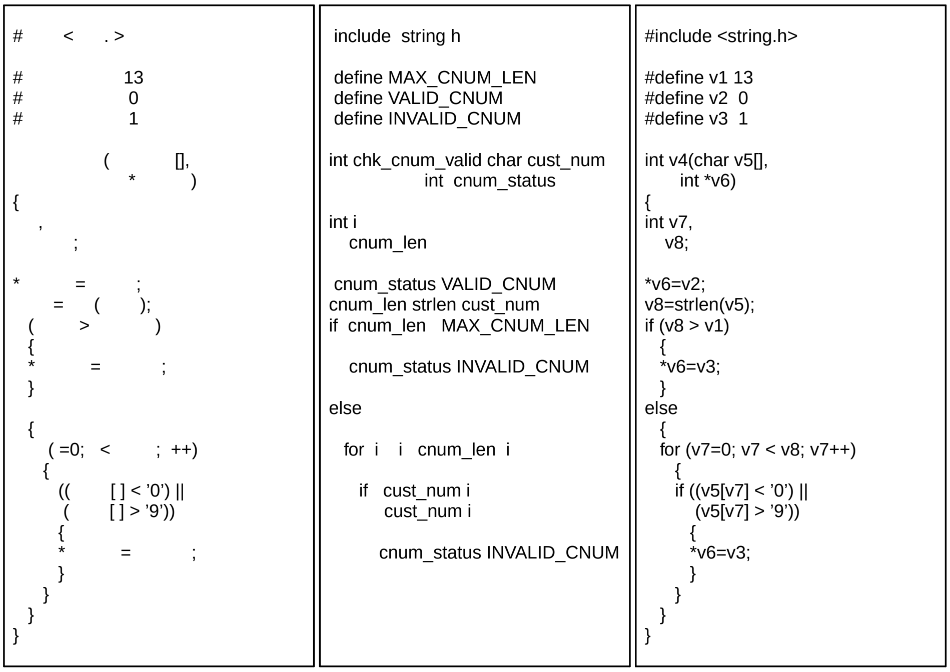
A study by Martinez and Monperrus[1200] investigated the kind of changes made to source to fix reported faults. The top five changes accounted for 30% of all changes, i.e., add method call, add `if-statement`, change method call, delete method call, and delete `if-statement`. Figure 7.27 shows kind of source changes ranked by percentage occurrence, and exponentials fitted over a range of ranks (red lines).

7.2.8 Identifier names

Identifier names provide a channel through which the person writing the code can communicate information to subsequent readers. The communications channel operates through the semantic associations triggered in the mind of a person as they read the source (tools might also attempt to gather and use this semantic information).

Semantic associations may be traceable to information contained in the source (e.g., the declared type of an identifier), or preexisting cultural information present in writers' or readers' memory (e.g., semantic interpretations associated with words in their native language within the culture it was learned and used).

Given that most functions are only ever modified by the original author (see fig 7.16), the primary beneficiary of any investment in naming of local identifiers is likely to be the developer who created them.



```
#     <    . >

#              13
#              0
#              1

           (          [],
               *          )
{
   ,
        ;

*        =         ;
     =    (      );
  (       >          )
  {
  *        =          ;
  }

  {
   ( =0;  <        ; ++)
   {
    ((       [ ] < '0') ||
     (       [ ] > '9'))
    {
    *        =          ;
    }
   }
  }
}
```

```
 include  string h

 define MAX_CNUM_LEN
 define VALID_CNUM
 define INVALID_CNUM

int chk_cnum_valid char cust_num
                 int  cnum_status

int i
    cnum_len

 cnum_status VALID_CNUM
cnum_len strlen cust_num
if  cnum_len   MAX_CNUM_LEN

    cnum_status INVALID_CNUM

else

   for  i    i   cnum_len  i

      if   cust_num i
           cust_num i

         cnum_status INVALID_CNUM
```

```
#include <string.h>

#define v1 13
#define v2  0
#define v3  1

int v4(char v5[],
       int *v6)
{
int v7,
    v8;

*v6=v2;
v8=strlen(v5);
if (v8 > v1)
   {
   *v6=v3;
   }
else
   {
   for (v7=0; v7 < v8; v7++)
      {
      if ((v5[v7] < '0') ||
          (v5[v7] > '9'))
         {
         *v6=v3;
         }
      }
   }
}
```

Figure 7.28: Three versions of the source of the same program, showing identifiers, non-identifiers and in an anonymous form; illustrating how a reader's existing knowledge of English word usage can reduce the cognitive effort needed to comprehend source code. Based on an example from Laitinen.[1057]

Identifiers are the most common token in source (29% of the visible tokens in `.c` files,[919] with comma the second most common at 9.5%), and they represent approximately 40% of all non-white-space characters in the visible source (comments representing 31% of the characters in `.c` files).

Each identifier appearing in the visible source is competing for developer cognitive resources. Identifiers having similar spellings, pronunciations, or semantic associations may generate confusion, resulting in mistaken interpretations being made; identifiers with long names may consume cognitive resources that are out of proportion to the benefits of the information they communicate. Figure 7.29 shows the number of function definitions containing a given number of occurrences of identifiers (blue/green), and of distinct identifiers (red).

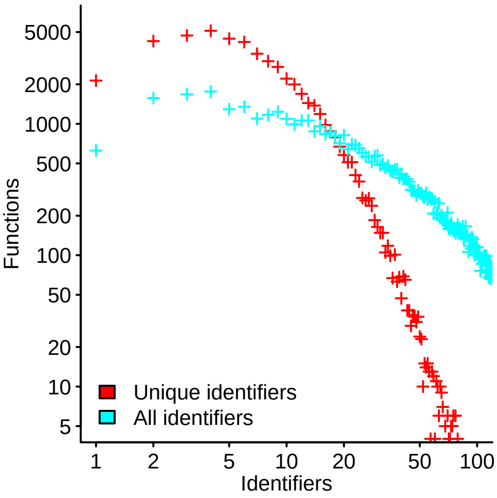


Figure 7.29: Number of C function definitions containing a given number of identifier uses (unique in red, all in blue/green). Data from Jones.[919] Github–Local

The meanings associated with a word, by a community, evolves,[1054] with different changes sometimes occurring in different geographic communities. One study[427] found people following a two-stage lifecycle: a linguistically innovative learning phase during which members align with the language of the community, followed by a conservative phase in which members don't respond to changes in community norms.

The same word may trigger different semantic associations in different people. For instance, the extent to which a word is thought to denote a concrete or abstract concept[1458] (*concrete* words, defined as things or actions in reality, experienced directly through the senses, whereas *abstract* words are not experienced through the senses, they are language-based with their meaning depending on other words). What is the probability that an identifier will trigger the same semantic associations in the mind of readers, when they encounter the identifier in code?

A study by Nelson, McEvoy and Schreiber[1343] investigated free association of words. Subjects were given a word, and asked to reply with the first word that came to mind. More than 6,000 subjects producing over 700,000 responses to 5,018 stimulus words.

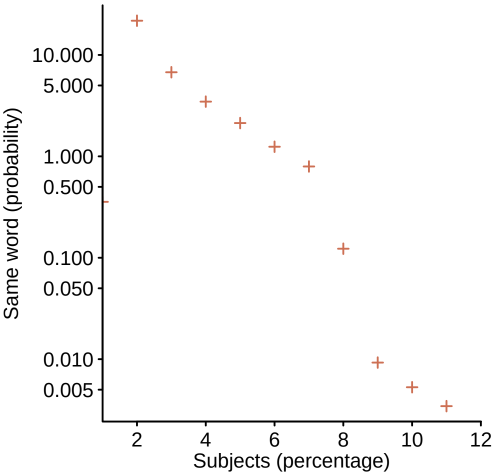


Figure 7.30: Probability (averaged over all cue words) that, for a given cue word, a given percentage of subjects will produce the same word. Data from Nelson et al.[1343] Github–Local

What is the probability that the same response word will be given by more than one subject? Figure 7.30 shows the probability (averaged over all words) that a given percentage of subjects will give the same word in response to the same cue word (values were calculated for each word, for two to ten subjects, and normalised by the number of subjects responding to that word). The mean percentage of unique responses was 18% (sd 9).

In this study subjects were not asked to think about any field of study and were mostly students (i.e., were not domain experts). Domain experts may be more likely to agree on a response, for terms specific to their domain.

Table 7.5 shows the percentage of identifiers occurring in each pair of seven large software systems; top row is the total number of identifiers in the visible source of each system.

Identifiers do not appear in isolation, in source, they appear within the context of other code. One study[921] found that identifier names can have a large impact on decisions made

	gcc	idsoftware	linux	netscape	openafs	openMotif	postgresql
identifiers	46,549	27,467	275,566	52,326	35,868	35,465	18,131
gcc	-	2	9	6	5	3	3
idsoftware	5	-	8	6	5	4	3
linux	1	0	-	1	1	0	0
netscape	5	3	8	-	5	7	3
openafs	6	4	12	8	-	3	5
openMotif	4	3	6	11	3	-	3
postgresql	9	5	12	11	10	6	-

Table 7.5: Percentage of identifiers in one program having the same spelling as identifiers occurring in various other programs. First row is the total number of identifiers in the program, and the value used to divide the number of shared identifiers in that column. Data from Jones.[919]

about the relative precedence of binary operators in an expression. Also, naming inconsistencies between the identifier passed as an argument, and the corresponding parameter has been used to find coding mistakes.[1556]

A study[1656] of word choice in a fill-in-the-blank task (e.g., “in tracking the . . . ”), found that probability of word occurrence (derived from large samples of language use) was a good predictor of both the words chosen, and the order in which subjects produced them (subjects were asked to provide 20 responses per phrase).

English pronunciation can be context dependent, e.g., “You can lead a horse to water, but a pencil must be lead.” and “Wind the clock when the wind blows.”

Speakers of different natural languages will have trained on different inputs, and during school people study different subjects (each having its own technical terms). A study by Gardner, Rothkopf, Lapan, and Lafferty[644] asked subjects (10 engineering, 10 nursing, and 10 law students) to indicate whether a letter sequence was a word or a nonword. The words were drawn from a sample of high frequency words (more than 100 per million), medium-frequency (10–99 per million), low-frequency (less than 10 per million), and occupationally related engineering or medical words.

The results showed engineering subjects could more quickly and accurately identify the words related to engineering (but not medicine); the nursing subjects could more quickly and accurately identify the words related to medicine (but not engineering). The law students showed no response differences for either group of occupationally related words. There were no response differences on identifying nonwords. The performance of the engineering and nursing students on their respective occupational words was almost as good as their performance on the medium-frequency words.

Object naming has been found to be influenced by recent experience,[1704] practical skills (e.g., typists selecting pairs of letters that they type using different fingers[1852]) and egotism (e.g., a preference for letters in one’s own name or birthday related numbers[1001, 1377]).

Developers may select the same identifier for different reasons. A study[169] of the use of single letter identifiers in five languages found that `i` was by far the most common in source code written in four of the languages. This choice might be driven by abbreviating the words `integer` (the most common variable type) or `index`, or by seeing this usage in example code in books and on the web, or because related single letters were already used.

Desirable characteristics in an identifier name include: high probability of triggering the appropriate semantic associations in the readers’ mind, a low probability of being mistaken for another identifier present in the associated source code, and consuming cognitive resources proportional to the useful information it is likely to communicate to readers.

Studies of the characteristics of words, in written form, found to have an effect on some aspect of subject performance include: *word length effect*,[1348] age of acquisition[398, 1626] (when the word was first learned), frequency of occurrence[95] (e.g., in spoken form and various kinds of written material), articulatory features of the initial phoneme (listener analysis of a spoken word is initiated when the first sounds are heard; differences at the beginning enable distinct words to be distinguished sooner).

Most studies of words have investigated English, but a growing number of large scale studies are investigating other languages.[589] Orthography (the spelling system for the written form of a language) can have an impact on reader performance, English has a deep orthography (i.e., a complex mapping between spelling and sound), while Malay has a shallow orthography (i.e., a one-to-one mapping between spelling and sound; also

Spanish), and word length in Malay has been found to be a better predictor of word recognition than word frequency.[1971]

When creating a spelling for an identifier, a path of least cognitive effort is for developers to rely on their experience of using their own native language, e.g., lexical conventions, syntax,[295] word ordering conventions (adjectives). The mistakes made by developers, in the use of English, for whom English is not a native language are influenced by the characteristics of their native language.[1781]

What are the characteristics likely to increase the likelihood that an identifier will be mistaken for another one?

A study by Lambert, Chang, and Gupta[1060] investigated drug name confusion errors.[ix] Subjects briefly saw the degraded image of a drug name. Both the frequency of occurrence of drug names, and their neighborhood density were found to be significant factors in subject error rate.

An analysis of the kinds of errors made found that 234 were omission errors and 4,128 were substitution errors. In the case of the substitution errors, 63.5% were names of other drugs (e.g., Indocin® instead of Indomed®), with the remaining substitution errors being spelling-related or other non-drug responses (e.g., Catapress instead of Catapres®).

Dearest creature in creation,
Study English pronunciation.
I will teach you in my verse
Sounds like corpse, corps, horse, and worse.
I will keep you, Suzy, busy,
Make your head with heat grow dizzy.
Tear in eye, your dress will tear.
So shall I! Oh hear my prayer.
Pray, console your loving poet,
Make my coat look new, dear, sew it!

Just compare heart, beard, and heard,
Dies and diet, lord and word,
Sword and sward, retain and Britain.
(Mind the latter, how it's written.)
Now I surely will not plague you
With such words as plaque and ague.
But be careful how you speak:
Say break and steak, but bleak and streak;
Cloven, oven, how and low,
Script, receipt, show, poem, and toe.

THE CHAOS (first two verses)
by Dr. Gerard Nolst Trenité, 1870-1946

Identifiers often contain character sequences that do not match words in the native language of the reader. Studies of prose have included the use non-words, often as a performance comparison against words, and nonwords are sometimes read as a word whose spelling it closely resembles.[1495]

Studies of letter similarity have a long history,[1816] and tables of visual[1304] (e.g., `1` (one) and `l` (ell)) and acoustic[1460] letter confusion have been published. When categorizing a stimulus, people are more likely to ignore a feature than they are to add a missing feature, e.g., **Q** is confused with **O** more often than **O** is confused with **Q**.

A study by Avidan[93] investigated how look it took subjects to work out what a Java method did, recording the time taken to reply. In the control condition subjects saw the original method, and in the experimental condition the method name was replaced by xxx, with local and/or parameter names replaced by single letter identifiers; in all experimental conditions the method name was replaced by xxx.

Subjects took longer to reply for the modified methods. When parameter names were left unmodified, subjects were 268 seconds slower (on average), and when locals were left unmodified 342 seconds slower (the standard deviation of the between subject differences was 187 and 253 seconds, respectively); see Github–sourcecode/Avidan-MSc.R.

A study[302] of the effectiveness of two code obfuscation techniques (renaming identifiers and complicating the control flow) found that renaming identifiers had the larger impact on the time taken by subjects to comprehend and change code; see Github–sourcecode/AssessEffectCodeOb.R.

One study[832] found that the time taken to find a mistake in a short snippet of code was slightly faster when the identifiers were words (rather than non-words); see Github–sourcecode/shorter-iden.R.

The names of existing identifiers are sometimes changed.[70] The constraints on identifier renaming include the cost of making all the necessary changes in the code and dependencies other software may have on existing names (e.g., identifier is in a published API).

One study[1500] created a tool that learned patterns of identifier usage in Javascript, which then flagged identifiers whose use in a given context seemed unlikely to be correct.

7.2.9 Programming languages

Thousands of programming languages have been created, and new languages continue to be created (see section 4.6.1); they can be classified according to various criteria.[1861]

Do some programming languages require more, or less, effort from developers, to write code having any of the desirable characteristics discussed in this chapter?

Every programming language has its fans, people who ascribe various positive qualities to programs written in this language, or in languages supporting particular characteristics.

[ix] Errors involving medication kill one person every day in the U.S., and injure more than one million every year; confusion between drug names that look and sound alike account for 15% to 25% of reported medication errors

There is little or no experimental evidence for any language characteristics having an impact on developer performance, and even less evidence for specific language features having a performance impact.

The term *strongly typed* is applied as a marketing term to languages believed by the speaker to specify greater than some minimum amount of type checking. The available experimental evidence for the possible benefits of using strongly typed languages is discussed in section 7.3.6. Languages provide functionality and developers can choose to make use of it, or not. It would be more appropriate to apply the term strongly typed to source code that takes full advantage of the type checking functionality provided by a language.

There is often a mechanism for subverting a language's built-in type checks, e.g., the use of **unconstrained** in Ada, the `unsafe` package in Go,[400] and the **unsafe** keyword in Rust.[555]

Factors that might generate a measurable difference in developer performance, when using different programming languages, include: individual knowledge and skill using the language, and interaction between the programming language and problem to be solved, e.g., it may be easier to write a program to solve a particular kind of problem using language X than using language Y.

Studies[1334, 1857, 2010] that compare languages using small programs suffer from the possibility of a strong interaction between the problem being solved, the available language constructs (requiring a sample containing solutions to a wide variety of problems), and the developers' skill at mapping the problem constructs available in the language used. Some languages include support for programming in the large (e.g., sophisticated separate compilation mechanisms), and studies will need to use large programs to investigate such features.

A study by Back and Westman[102] investigated the characteristics of the 220,349 entries submitted to the Google code jam program competition for the years 2012-2016 (a total of 127 problems). Figure 7.31 shows the number of solutions containing a given number of lines for one of the problems (the one having the most submitted solution: 2,624), stratified by the five most commonly used languages.

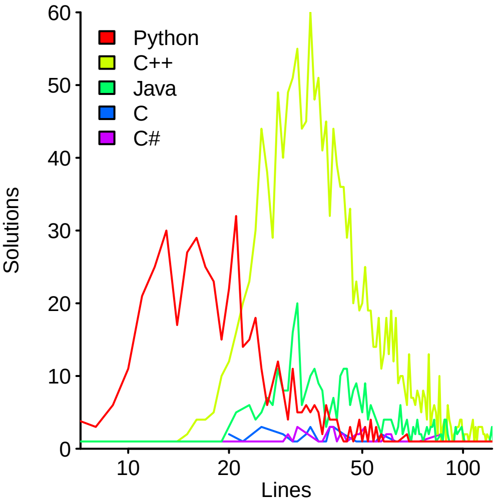


Figure 7.31: Number of solutions to one a problem posed in a Google code jam competition, containing a given number of lines, stratified by programming language. Data from Back et al.[102] Github–Local

A realistic comparison of two languages requires information from many implementations of large systems targeting the same application domain.

A study by Waligora, Bailey and Stark[1897] compared various lines of code measurements (e.g., measurements of declarations, executable statements and code reuse) of 11 Ada and 19 Fortran implementations of NASA ground station software systems. While there were differences in patterns of behavior for various line counts, these differences could have been caused by factors outside the scope of the report (e.g., the extent to which individual companies were involved in multiple projects and in a good position to evaluate the potential for reuse of code from previous projects, or the extent to which project requirements specified that code should be written in a way likely to make it easier to reuse); see Github–projects/nasa-ada-fortran.R.

Differences in performance between subjects, and learning effects, can dominate studies based on small programs, or experiments run over short intervals. It is possible to structure an experiment such that subject performance improvement, on each reimplementation (driven by learning that occurred on previous implementations), is explicitly included as a variable; see section 11.6.

A study by Savić, Ivanović, Budimac and Radovanović[1614] investigated the impact of a change of teaching language on student performance in practical sessions (going from Modula-2 to Java). Student performance, measured using marks assigned, was unchanging across the four practical sessions, as was mean score for each year; see Github–ecosystems/2016-sclit-uup.R for details.

A study by Prechelt and Tichy[1505] investigated the impact of parameter checking of function calls (when the experiment was performed, C compilers that did not perform any checking on the arguments passed to functions, so-called K&R style, were still in common use). All subjects wrote two programs: one program using a compiler that performed argument checking of function calls, and the second program using a compiler that did not perform this checking. Subjects were randomly assigned to the problem to solve first, and the compiler to use for each problem. The time to complete a correct program was measured.

Fitting a regression model to the results shows that the greatest variation in performance occurred between subjects (standard deviation of 74 minutes), the next largest effect was problem ordering (with the second problem being solved 38 minutes, on average, faster than the first). The performance improvement attributed to argument checking is 12 minutes, compared to no argument checking; see Github–experiment/tcheck98.R.

Studies[1269] investigating the use of a human language, to specify solutions to problems, have found that subjects make extensive use of the contextual referencing that is common in human communication. This use of context, and other issues, make it extremely difficult to automatically process the implementation instructions.

Programming languages that support coding constructs at a level of abstraction higher than machine code have to make some implementation decisions about lower level behavior, e.g., deciding the address of variables defined by the developer, and the amount of storage allocated to hold them. These implementation decisions are implicit behavior.

Studies[1050] have investigated the use of particular kinds of implicit behavior, and fault repositories provide evidence that some coding mistakes are the result of developers not understanding the implicit behavior present in a particular sequence of code.

Your author is not aware of any evidence-based studies showing that requiring all behavior to be explicit, in the code, is more/less cost effective (i.e., supporting implicit behavior is less/more costly than the cost of [the assumed] more coding mistakes). Neither is your author aware of any evidence-based studies showing that alternative implicit behavior result in fewer developer mistakes.

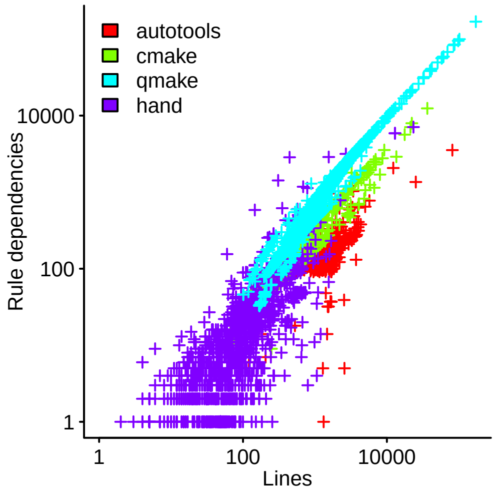


Figure 7.32: Number of lines against number of dependencies contained in rules, in 19,689 makefiles, stratified by method of creation. Data from Martin.[1197] Github–Local

7.2.10 Build bureaucracy

Software systems are built by selectively combining source code, contained in multiple files, libraries, and data files. Some form of bureaucracy is needed to keep track of the components required to perform a build, along with any dependencies they have on other components, and the tools (plus appropriate options) used to map the input files to the desired final software system. Build systems that have been created include: tools that process rules specifying operations to be performed (e.g., `make` operating on makefiles), and tools that create files containing target specific rules from higher level requirements (e.g., a `configure` script generates the makefiles appropriate to the build system). Also see section 5.4.7.

A study by Martin[1197] investigated the features used in 19,689 makefiles. Figure 7.32 shows the number of lines contained in these makefiles, along with the number of dependencies contained in the rules of the respective file. Most of the larger files have been generated by various tools that process higher level specifications, with smaller files being mostly handwritten.

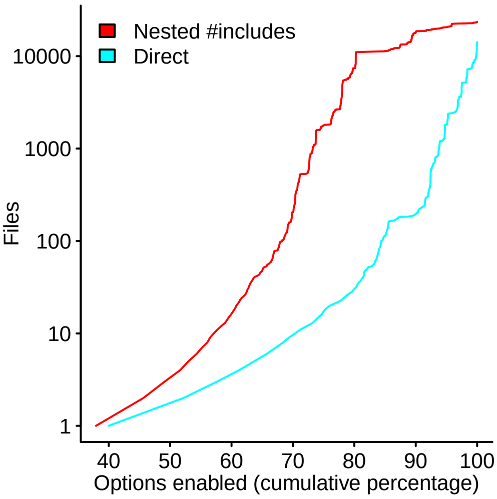


Figure 7.33: Cumulative percentage of configuration options impacting a given number of source files in the Linux kernel. Data kindly provided by Ziegler.[2006] Github–Local

Program source code may be written in a way that supports optional selection of features at build time. One technique for selecting the code to process during compilation is conditional compilation, e.g., `#ifdef/#endif` in C and C++ checks whether an identifier is defining, or not (the identifier is sometimes known as a *feature test macro*, *feature constant*, or *build flag*).

A study by Liebig, Apel, Lengauer, Kästner and Schulze[1126] measured various attributes associated with the use of conditional compilation directives in 40 programs written in C (header file contents were ignored). Figure 7.34 shows the number of unique *feature constants* appearing in programs containing a given number of lines of code.

How extensive is the impact of build flags on source code? A study by Ziegler, Rothberg and Lohmann[2006] investigated the number of source files in the Linux kernel affected by configuration options. Figure 7.33 shows the number of files affected by the cumulative percentage of configuration options. The impact of 37.5% of options is restricted to a single file, and some options have an impact over tens of thousands of files.

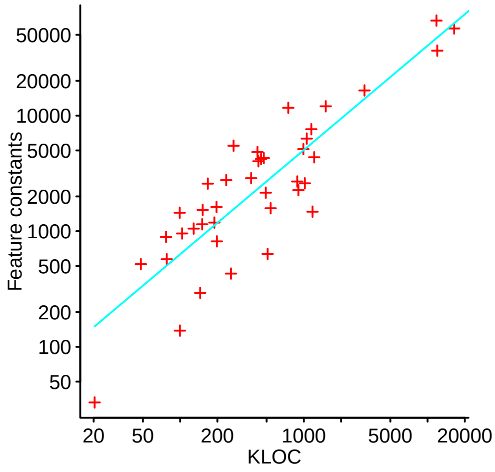


Figure 7.34: Number of *feature constants* against LOC for 40 C programs, with fitted regression line. Data from Liebig et al.[1126] Github–Local

Applications may be shipped with new features that are not fully tested, or that may sometimes have undesirable side effects. User accessible command line (or configuration file) options may be used to switch features on/off. A study by Rahman, Shihab and Rigby[1535] investigated the feature toggles supported by Google Chrome. The code supporting a given feature may be scattered over multiple files and provides an insight into the organization of program source. Figure 7.35 shows a density plot of the number of files involved in each feature of Google Chrome; the number of feature toggles grew from 6 to 34 over these four releases.

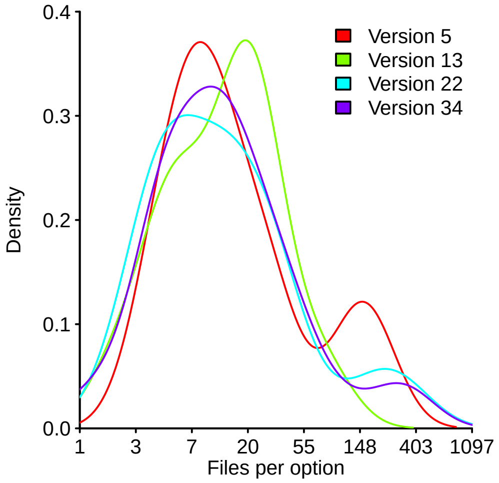


Figure 7.35: Density plot of the number of files containing code involved in supporting distinct options in four versions of Google Chrome. Data from Rahman et al.[1535] Github–Local

As source code evolves the functionality provided by a package or library may cease to be used, removing the dependency on this package or library. A missing dependency is likely to be flagged at build time, but unnecessary dependencies are silently accepted. The result is that over time the number of unnecessary, or redundant, dependencies grows.[1717]

One study[1992] of C/C++ systems found that between 83% and 97% recompilations, specified in makefiles, were unnecessary.

7.2.11 Folklore metrics

Two source code metrics, proposed in the 1970s, have become established folklore within many software development communities: Halstead's metric and McCabe's cyclomatic complexity metric. Use of these metrics persists, despite the studies[891, 1065, 1178] finding that they have predictive performance that is comparable to that of lines of code, when used to model various program characteristics. Why do developers and researchers continue to use them? Perhaps counting lines of code is thought to lack the mystique that the market craves, or is the demand for more accurate metrics insufficient to motivate a search for something better?

Ptolomy's theory (i.e., the Sun and planets revolved around the Earth) was not discredited because it gave inaccurate answers, but because Copernicus's theory eventually[x]made predictions that had a better fit to experimental measurements (once observing equipment became more accurate).

The studies involving what became known as Halstead's complexity metric were documented by Halstead in a series of technical reports[272, 627, 706, 763, 764, 1408] published in the 1970s. The later reports compared theory with practice, using tiny datasets; Halstead's early work[764] is based on experimental data obtained for other purposes,[2010] and later work[627] used a data set containing nine measurements of four attributes (see Github–faults/halstead-akiyama.R), others contained 11 measurements,[706] or used measurements from nine modules.[1408]

Halstead gave formula for what he called *program length* (with source code tokens as the unit of measurement), *difficulty*, *volume* and *effort*. A dimensional analysis shows that the first three each have units of *length*, and *effort* has units of *area* (i.e., *length*^{2}), i.e., the names suggest that different quantities are being calculated, but they all calculate the same quantity in different ways.

Researchers at the time found that Halstead's metric did not stand up to scrutiny: Magidin and Viso[1178] used slightly larger datasets (50 randomly selected algorithms), and found many mistakes in the methodology used in Halstead's papers; a report by Shen, Conte and Dunsmore,[1667] Halstead's colleagues at Purdue, written four years after Halstead's flurry of papers, contains a lot of background material, and points out the lack of any theoretical foundation for some of the equations, that the analysis of the data was weak, and that a more thorough analysis suggests theory and data don't agree.

The Shen et al report explains that Halstead originally proposed the metrics as a way of measuring the complexity of algorithms not programs, explains the background to Halstead's uses of the term *impurities*, and the discussion for the need for *purification* in his early work. Halstead points out[272] that the value of metrics for *algorithms written by students* are very different from those for the equivalent programs published in journals, and goes on to list eight classes of impurity that need to be purified (i.e., removing or rewriting clumsy, inefficient or duplicate code) in order to obtain results that agree with the theory (i.e., cleaning data in order to obtain results that more closely agree with his theory).

A study by Israeli and Feitelson[891] investigated the evolution of the Linux kernel. Figure 7.36 shows lines of code against Halstead's volume and McCabe's cyclomatic complexity, for the 62,365 C functions containing at least 10 lines, in Linux version 2.6.9. Explanations for the value of the exponents of the fitted power laws include: for Halstead's volume: the number of tokens per line increases with function size, and for McCabe's complexity: that the number of control structures decreases as function size increases.

The paper in which McCabe[1213] proposed what he called *complexity measures* contains a theoretical analysis of various graph-theoretic complexity measures, and some wishful thinking that these might apply to software; no empirical evidence is given. Studies[1065]

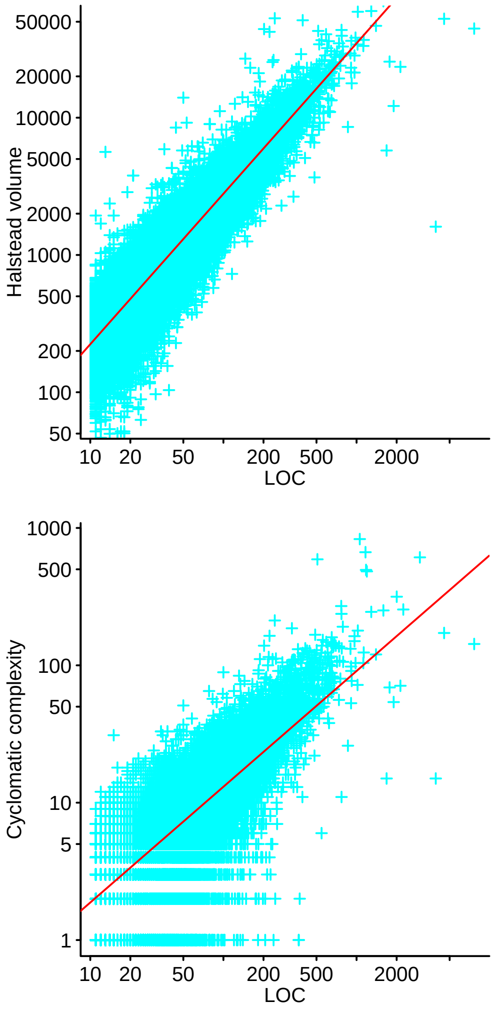


Figure 7.36: Lines of code, Halstead's volume and McCabe's cyclomatic complexity of the 62,365 C functions containing at least 10 lines, in Linux version 2.6.9; fitted regression lines have the form: $Halstead\_volume \propto KLOC^{1.1}$ and $McCabe\_complexity \propto KLOC^{0.8}$. Data from Israel et al.[891] Github–Local

[x] Thanks to Isaac Newton.

have found a strong correlation between lines of code and McCabe's complexity metric. This metric also suffers from the problem of being very easy to manipulate (i.e., is susceptible to software accounting fraud; see section 6.4.2).

7.3 Patterns of use

Patterns of source code use are of general interest to the extent they provide information that aids understand the software development process. Common usage patterns have niche interest groups, such as compiler writers wanting to maximise their investment in code optimizations by focusing on commonly occurring constructs, by static analysis tools focusing on the mistakes commonly made by developers, and by teachers looking to minimise what students have to know to be able to handle most situations (and common novice mistakes[912]).

Patterns that occur during program execution[1558] can be used to help tune the performance of both the measured program and any associated runtime system; effort might also be focused on individual language constructs.[1557]

The spoken form of human languages have common patterns of word usage[1094] and phrase usage,[189] the written form of languages also have common patterns of letter usage.[930] Common usage patterns are also present in the use of mathematical equations.[1710]

A theory is of no use unless it can be used to make predictions that can be verified (or not), and any theory of developer coding behavior needs to make predictions about detectable patterns in code. For instance, given the hypothesis that developers are more likely to create a separate function for heavily nested code, then the probability of encountering an `if-statement` should decrease with increasing nesting depth.[xi]

Common usage patterns in human written source code are driven by developer habits (perhaps carried over from natural language usage, or training material), recurring patterns of behavior in the application domain, hardware characteristics, the need to interface to code written by others, or influenced by the characteristics of the devices used to write code. For instance, the width of computer screens limits the number of characters visible on a line within a window. Figure 7.37, upper plot, shows the number of lines, in C source files, containing a given number of characters. The sharp decline in number of lines occurs around a characters-per-line values supported by traditional character based terminals.[xii]

Some code may be automatically generated. Figure 7.37, lower plot, shows the number of C `selection-statements`[xiii] occurring at a given maximum nesting depth. One interpretation of the decreasing trend, at around a nesting level of 13, is that automatically generated code becomes more common at this depth, than human written code.

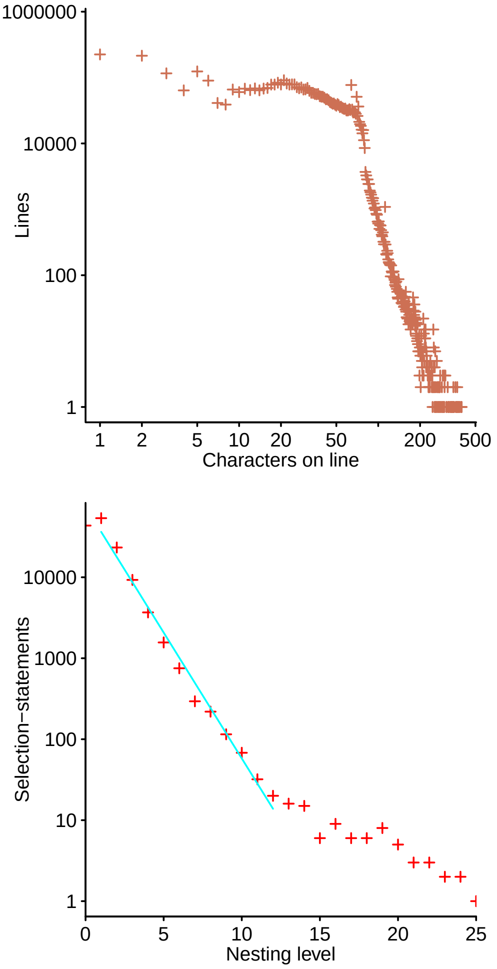


Figure 7.37: Number of `selection-statements` having a given maximum nesting level; fitted regression line has the form: $num\_selection \propto e^{-0.7nesting}$. Data from Jones.[919] Github–Local

Common patterns may exist because a lot of code is built from sequences of simple building blocks (e.g., assigning one variable to another), and there are a limited number of such sequences (particularly if the uniqueness of identifiers is ignored).

A study by Lin, Ponzanelli, Mocci, Bavota and Lanza[1133] investigated the extent to which the same sequence of tokens (as specified by the Java language, with identifiers having different names treated as distinct tokens) occurred more than once in the source of a project. Figure 7.38 shows violin plots of the fraction of a project's token sequences of a given length (for sequence lengths between 3 and 60 tokens) that appeared more than once in the projects Java source (for each of 2,637 projects).

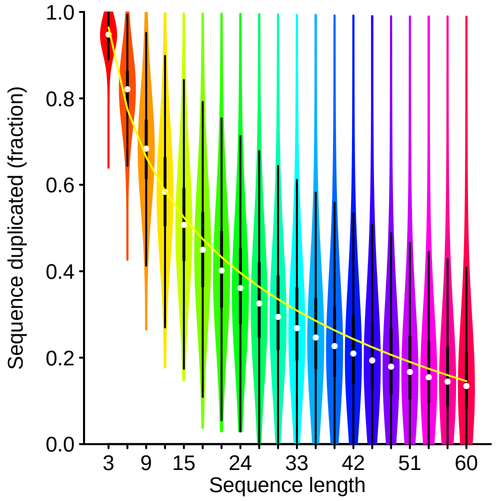


Figure 7.38: Fraction of a project's token sequences, containing a given number of tokens, that appear more than once in the projects' Java source (for 2,637 projects); the yellow line has the form: $fraction \propto a - b * \log(seq\_len)$, where$a$ and b are fitted constants. Data from Lin et al.[1133] Github–Local

A study by Baudry, Allier and Monperrus[142] investigated the possibility of generating slightly modified programs (e.g., add, replace and delete a statement) having the same behavior (i.e., passing the original program's test suite; the nine large Java programs used had an average statement coverage of 85%). On average, 16% of modified programs produced by the best performing generated add-statement algorithm passed the test suite; 9% for best performing replace and delete; see Github–sourcecode/Synth-Diverse-Programs.csv.xz.

Individual developers have personal patterns of coding usage.[285] These coding accents are derived from influences such as developer experience with using other languages, ideas picked up from coding techniques appearing in books, course notes, etc.

[xi] Which does not occur in practice, at least in C code.

[xii] Historically, typewriters supported around 70 characters per line, depending on paper width, and punched cards supported 80 characters.

[xiii] `if-statements` and `switch-statements`.

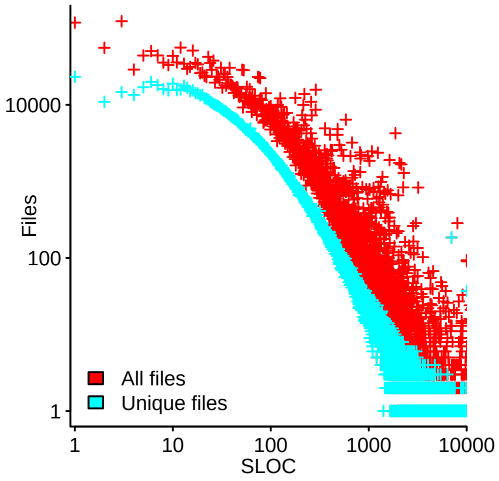


Figure 7.39: Number of Python source files containing a given number of SLOC; all files, and with duplicates removed. Data from Lopes et al.[1145] Github–Local

There may be common patterns specific to application domains, programming language or large organizations. The few existing studies have involved specific languages in broad domains (e.g., desktop computer applications written in C[919] and Java usage in open source repositories[726]), and without more studies it is not possible to separate out the influences driving the patterns found.

A variety of practices can bias into source code measurement data, including:

- when making use of source written by third-parties, it may be more cost effective to maintain a local copy of the source files, than link to the original. A consequence of this behavior is the presence of duplicate source files in public repositories (skewing population measurements), and individual project measurements may be unduly influenced by the behavior of developers working on other projects.

  A study by Lopes, Maj, Martins, Saini, Yang, Zitny, Sajnani and Vitek[1145] investigated duplicate code in 4.5 million non-forked Github hosted projects (i.e., known forks of a project were not included in the analysis). Of the 428+ millions source files written in Java, C++ , Python or Javascript, 85 million were unique. Table 7.6 shows the percentage of unique files by language, and percentage of projects containing at least a given percentage of duplicates files.

	Java	C++	Python	JavaScript
Unique files	60%	27%	31%	7%
Projects	1,481,468	364,155	893,197	1,755,618
duplicates > 50%	14%	25%	18%	48%
duplicates > 80%	9%	16%	11%	31%
duplicates 100%	6%	7%	6%	15%

Table 7.6: Percentage of unique files for a given language, number of projects, and average percentage of duplicated files in projects for Github hosted projects written in various languages. Data from Lopes et al.[1145]

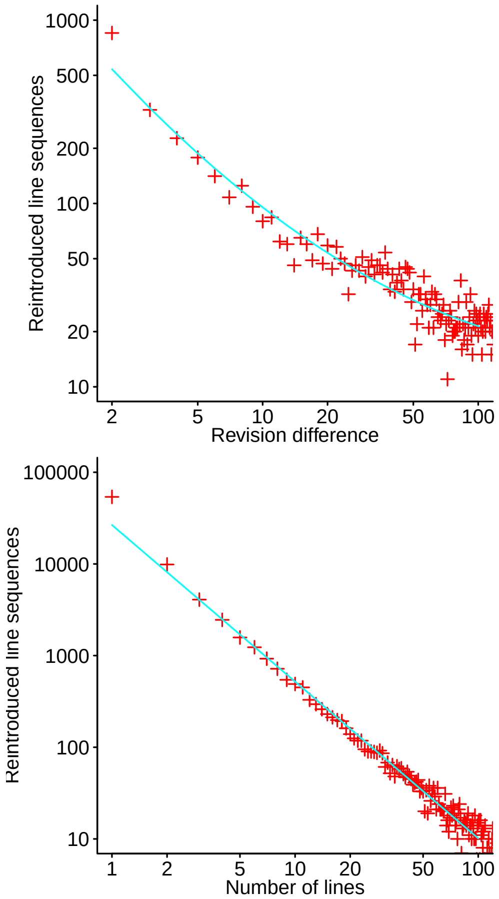


Figure 7.40: Number of reintroduced line sequences having a given difference in revision number between deletion and reintroduction (upper), and number of reintroduced line sequences containing a given number of lines (lower); the fitted regression lines have the form: $Occurrence \propto NumLines^{-1.4}e^{0.1\log(NumLines)^2}$ and $Occurrences \propto NumLines^{-1.7}$. Data kindly provided by Kamiya.[955] Github–Local

  Figure 7.39 shows the number of Python files containing a given number of SLOC, for a 10% sample of all 31,602,780 files, and the 9,157,622 unique files,

- the decision about which code matches a particular pattern may not be a simple yes/no, but involve a range, e.g., the number of tokens needing to match before a code sequence is considered to be a clone. Tools used to extract patterns from source code often provide options for controlling their behavior,[1533]

- when modifying source, some developers commit every change they make to the project-wide repository, while other developers only make project-wide commits of code they have tested (i.e., they commit changes of behavior, not changes of code).[1342]

  When every change is committed, there will be more undoing of previous changes, than when commits are only made after code has been tested.

  A study by Kamiya[955] investigated how many deleted lines of code were added back to the source in a later revision, in a FreeBSD repository of 190,000 revisions of C source. Figure 7.40, upper: shows the number of reintroduced line sequences having a given difference in revision number between deletion and reintroduction, and lower: number of reintroductions of line sequences containing a given number of lines, with fitted power laws.

7.3.1 Language characteristics

The idea that the language we use influences our thinking is known as the *Sapir-Whorf* or *Whorfian* hypothesis.[659] The *strong language-based* view is that the language used influences its speakers' conceptualization process; the so-called *weak language-based* view is that linguistic influences occur in some cases, such as the following:

- *language-as-strategy*: language affects speakers performance by constraining what can be said succinctly with the set of available words; a speed/accuracy trade-off, approximating what needs to be communicated in a brief sentence rather than using a longer sentence to be more accurate,[867]

- *thinking-for-speaking*: for instance, English uses count nouns, which need to modified to account for the number of items, which requires speakers to pay attention to whether one item, or more than one item, is being discussed; Japanese nouns make use of classifiers, e.g., shape classifiers such as hon (long thin things) and mai (flat things), and measuring classifiers such as yama (a heap of) and hako (a box of). Some languages assign a gender to object names, e.g., the Sun is feminine in German, masculine in Spanish and neuter in Russian.

thinking for coding occurs when creating names for identifiers, where plurals may sometimes be used (e.g., the `rowsum` and `rowSums` functions in `R`),

- languages vary in the way they articulate numbers containing more than one digit, e.g., the components contained in 24 might be ordered as $20+4$ (English) or $4+20$ (French). Linguistic influences on numerical cognition have been studied.[264]

While different languages make use of different ways of describing the world, common cross-language usage patterns can be found. A study by Berlin and Kay[180] isolated what they called the *basic color terms* of 98 languages. They found that the number and kind of these terms followed a consistent pattern, see figure 7.41; while the boundaries between color terms varied, the visual appearance of the basic color terms was very similar across languages. Simulations of the evolution of color terms[130] suggest that it takes time for the speakers of a language to reach consensus on the naming of colors, and over time languages accumulate more color terms.

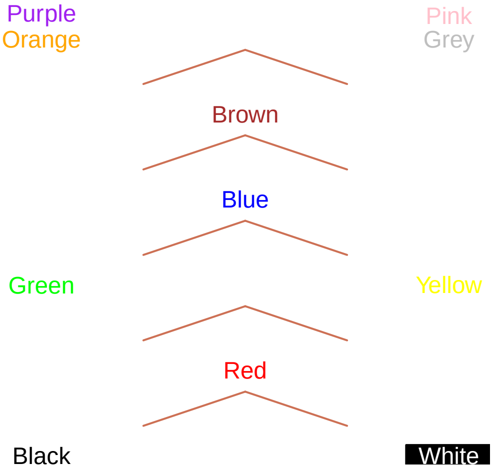


Figure 7.41: The Berlin and Kay[180] language color hierarchy. The presence of any color term in a language implies the existence, in that language, of all terms below it. Papuan Dani has two terms (black and white), while Russian has eleven (Russian may also be an exception in that it has two terms for blue.) Github–Local

Languages vary in their complexity, i.e., there is no mechanism that ensures all languages are equally complex.[1235]

Many programming languages in common use are still evolving, i.e., the semantics of some existing constructs are changing and support for new constructs is being added. Changing the semantics of language existing constructs involves a trade-off between alienating the developers and companies currently using the language (by failing to continue to process existing code), and fully integrating new constructs into the language.

At the end of 2008 the Python Software Foundation released Python 3, a new version of the language that was not compatible with Python 2. Over time features only available in Python 3 have been back-ported to Python 2. How have Python developers responded to the availability of two similar, but incompatible languages?

A study by Malloy and Power[1183] investigated the extent to which 50 large Python applications were compatible with various releases of the Python language. Figure 7.42 shows changes in the estimated mean compatibility of the 50 applications to 11 versions of Python, over time.

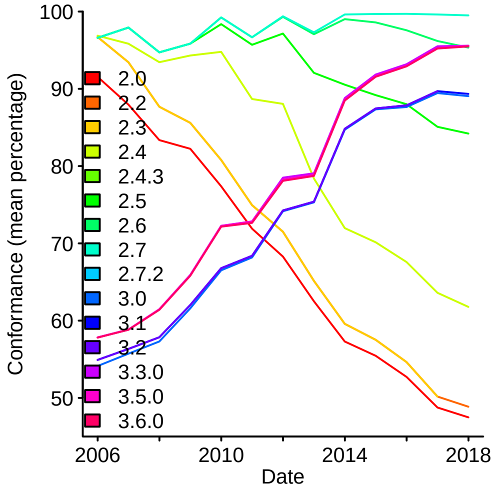


Figure 7.42: Mean compatibility of 50 applications to 11 versions of Python, over time. Data from Malloy et al.[1183] Github–Local

While new features are often added to languages, it is rare for a feature to be removed (at least without giving many years notice). The ISO Standards for both the Fortran and C have designated some constructed as deprecated, i.e., to be removed in the next release, but in practice they are rarely removed.[xiv]

Whatever the reason for the additions, removals, or modifications to a language, such changes will influence the characteristics of some of the code written by some developers. A new construct may add new functionality (e.g., atomics in C and C++), displace use of an existing construct (e.g., lambda expressions replacing anonymous classes[1209]).

How quickly are new languages constructs adopted, and regularly used by developers? Use of new language constructs depends on:

- compiler support. While vendors are quick to claim compliance with the latest standard, independent evidence is rarely available (e.g., compiler validation by an accredited test lab),[xv]
- existing developer knowledge and practices. What incentives do existing users of a language have to invest in learning new language features, and to then spend time updating existing habits? Is new language feature usage primarily driven by developers new to the language, i.e., learned about the new feature as a by-product of learning the language?.

[xiv] ANSI X3.9-1978,[63] known as Fortran 77, listed 24 constructs which it called conflicts with ANSI X3.9-1966. Some of these conflicts were removal of existing features (e.g., Hollerith constants), while others were interpretations of ambiguous wording. Subsequent versions of the Standard have not removed any language constructs.

[xv] When the British Standards Institute first offered C compiler validation, in 1991, three small companies new to the C compiler market paid for this service; all for marketing reasons. Zortech once claimed their C compiler was 100% Standard compliant (it was not illegal to claim compliance to a yet to be published standard still under development), and when the C Standard was published their adverts switched to claiming 99% compliance (i.e., a meaningless claim).

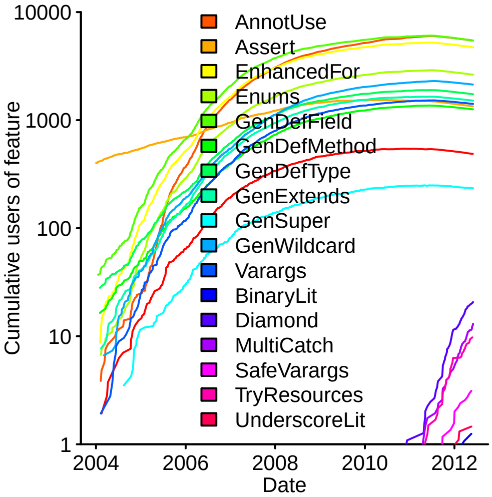


Figure 7.43: Cumulative number of developers who have committed Java source making use of particular new feature added to the language. Data from Dyer et al.[516] Github–Local

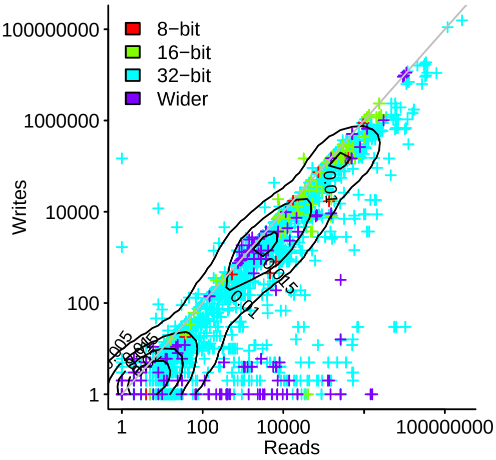


Figure 7.44: Number of reads and writes to the same variable, for 3,315 variables occupying various amounts of storage, made during the execution of the Mediabench suite; grey line shows where number of writes is the same as reads. Data kindly provided by Caspi.[298] Github–Local

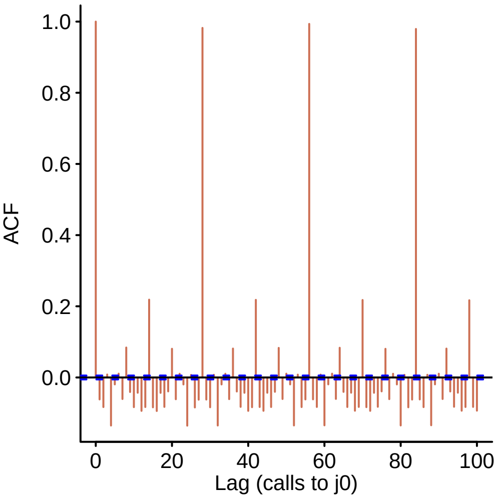


Figure 7.45: Autocorrelation function of the argument values passed to the Bessel function j0. Data kindly provided by Suresh.[1777] Github–Local

A handful of compilers now dominate the market for many widely used languages. The availability of good enough open source compilers has led to nearly all independent compiler vendors exiting the market.

For extensive compiler driven language usage to exist, widely used diverse compilers are required (something that was once common for some languages). With the small number of distinct compilers now in widespread use, any diversity of language construct use is likely to be driven by the evolution of compiler support for new language constructs, and the extent to which source has been updated to adapt to modified or new features.

A study by Dyer, Rajan, Nguyen and Nguyen[516] investigated the use of newly introduced Java language features, based on the source of commits made to projects on SourceForge. Figure 7.43 shows the cumulative growth (adjusted for the growth of registered SourceForge users[1976]) in the number of developers who had checked in a file containing a use of a given new feature, for the first time. Possible reasons for the almost complete halt in the growth of developers using a new Java language construct for the first time include: the emptying of the pool of developers willing to learn and experiment with new language features, and developers switching to other software hosting sites, e.g., Github became available in 2008.

A unit of source code may contain multiple languages, e.g., SQL,[54] assembler,[1564] or C preprocessor directives (also used in Fortran source to support conditional compilation), or database schema contained within string literals of the glue language used (such as PHP or Python[1135]).

7.3.2 Runtime characteristics

The runtime characteristics of interest to users of software, and hence of interest to developers, are reliability and performance. Reliability is discussed in chapter 6, and issues around the measurement of performance are discussed in chapter 13.

When execution time needs to be minimised, the relative performance of semantically equivalent coding constructs are of interest. A study by Flater and Guthrie[606] investigated the time taken to assign a value to an array element in C and C++ , using various language constructs. A fitted regression model contains interactions between almost every characteristic measured; see Github–benchmark/bnds_chk.R.

The interaction between algorithm used, size of data structures and hardware characteristics can have a large impact on performance, see fig 13.20.

Patterns of behavior that frequently occur during the execution of particular sequences of source code are of great interest to some specialists, and include (fig 7.5 illustrates that the relationship between static and dynamic behavior may have its own patterns).

- a symbiosis between cpu design and existing code; developers interested in efficiency attempt to write code that makes efficient use of cpu functionality, and cpu designers attempt to optimise hardware characteristics for commonly occurring instruction usage[27] and patterns of behavior (e.g., locality of reference[1280] can make it worthwhile caching previously used values, and the high degree of predictability of conditional branches, statically[122] and dynamically,[1281] can make it worthwhile for the cpu to support branch prediction),
- implementers of runtime libraries. Common patterns in the dynamic allocation of storage include: relatively small objects, with program-specific sizes make up most of the requests,[1193] once allocated the storage usually has a short lifetime,[1193] and objects declared using the same type name tend to have similar lifetimes.[1693]

  A study by Suresh, Swamy, Rohou and Seznec[1777] investigated the value of arguments passed to transcendental functions. Figure 7.45 shows the autocorrelation function of the argument values passed to the Bessel function j0.

7.3.3 Statements

Statements have been the focus of the majority of studies of source code; see fig 9.12 for percentage occurrence of various kinds of statements in C, C++ and Java.

Some languages support the creation of executable code at runtime, e.g., concatenating characters to build a sequence corresponding to an executable statement and then calling a function that interprets the string just as-if it had originally appeared in the source file.

A study by Rodrigues and Terra[1575] investigated the use of dynamic features in 28 Ruby programs. On average 2.6% of the language features appearing in the source were dynamic features; it was thought possible to replace 50% of the dynamic statements with static statements.

Figure 7.46 shows the number of dynamic statements, LOC, and methods appearing in Ruby programs containing a given number of dynamic constructs. Lines are a power law regression fit, with the exponents varying between 0.8 and 0.9.

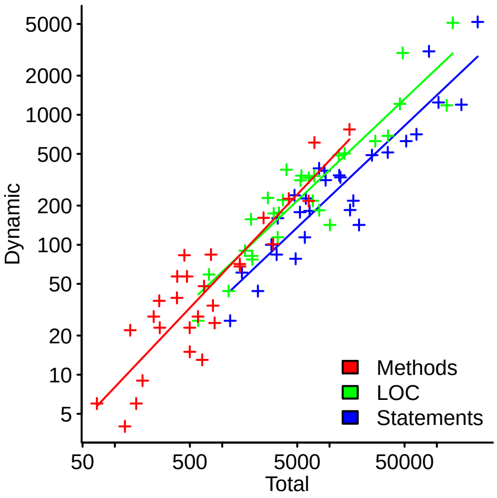


Figure 7.46: The number of dynamic statements, LOC and methods against total number of those constructs appearing in 28 Ruby programs; lines are power law regression fits. Data from Rodrigues et al.[1575] Github–Local

7.3.4 Control flow

An `if-statement` is a decision point, its use is motivated by either an application requirement or an internal house-keeping issue, e.g., an algorithmic requirement, or checking an error condition.

Developers often using indentation to visually delimit constructs contained within particular control flows; see fig 11.64.

The ordering of `if-statements` may be driven by developer beliefs about the most efficient order to perform the tests, as the code evolves adding new tests last, or other reasons. One study[1967] used profile information to reorder `if-statements`, by frequency of their conditional expression evaluating to true; the average performance improvement, on a variety of Unix tools, was 4%.

The control flow supported by many early programming languages closely mimicked the support provided by machine code.

The `goto-statement` was the work-horse of program flow control, and the term *spaghetti code* was coined to describe source code using `goto-statements` in a way that required excessive effort to untangle control flows through a program. The sometimes heated debates around the use of the `goto-statement`, from the late 1960s and 1970s,[492, 1017] have become embedded in software folklore, and continue to inform discussion, e.g., guidelines recommending against the use of `goto-statement`.[1274]

Over time higher level control flow abstractions have been introduced, e.g., *structured programming*; code in the margin shows an example of unstructured and structured code.

```
 if (a != 1)      if (a == 1)
    goto 100;        {
 b=1;                b=1;
 c=2;                c=2;
100:;                }
 d=3;             d=3;
```

A study by Osman, Leuenberger, Lungu and Nierstrasz[1406] investigated the use of checks against the null value in the `if-statements` within 800 Java systems. Figure 7.47 shows the number of `if-statements` against the number of these `if-statements` whose condition checks a variable against null.

Studies of the use of **goto** in Ada[657] and C[919] have found that it is mostly used to jump out of nested constructs, to statements appearing earlier or later; one study[1325] found that 80% of usage in C was related to error handling.

The lower plot in figure 7.37 shows the number of C `selection-statements`, occurring at a given maximum nesting depth. The probability of encountering a `selection-statement` remains relatively constant with nesting depth, implying that developers are not more likely to create a function to contain more deeply nested code, than any other code.

Many languages support a statement to handle the need to select one control flow path from multiple possibilities, based on the value of an expression, e.g., a `switch-statement`. Even when such a statement is available, developers may choose to use a sequence of `if-statements`. For instance, ordering the sequence to reflect the expected likelihood of the condition being true (in the belief that this improves performance); the code in the margin is from the source of a version of `grep`.

```
if ((c = *sp++) == 0)
   goto cerror;
if (c == '<') { ... }
if (c == '>') { ... }
if (c == '[') { ... }
if (c == ']') { ... }
if (c >= '1' && c <= '9') { ... }
```

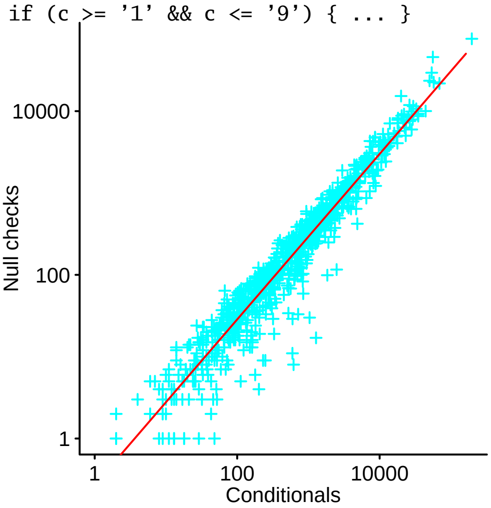


Figure 7.47: Total `if-statements` against `if-statements` whose condition involves a null check, in each of 800 Java projects; regression line fitted has the form: *null_checks* ∝ *Conditionals*. Data kindly provided by Osman.[1406] Github–Local

A study by Jones[922] investigated developer choice of control flow construct, when the problem allowed either `if-statement` or `switch-statement`, to be used. The questions involved writing a function that used the value of a parameter to select the value to assign to a specific variable; each question specified whether the parameter took 3, 4 or 5 values. The following shows two possible solutions to one question:

```
if (company == 1)
   X = "Intel";
else if (company == 20)
   y = "Motorola";
```

```
switch(company)
   {
   case 1: X = "Intel";
           break;
   case 20: y = "Motorola";
            break;
```

```
else if (company == 33)
   W = "IBM";
else if (company == 41)
   p = "Sun";
```

```
case 33: W = "IBM";
         break;
case 41: p = "Sun";
         break
}
```

A total of 199 questions were answered by 12 professional developers. One subject used the `if-else-if` form when the parameter contained three values, and a `switch-statement` when more than three values. Two subjects tended to always use the if-else-if form, and nine subjects always used an `switch-statement` (a few subjects answered one question using an `if-statement`).

A study by Durelli, Offutt, Li, Delamaro, Guo, Shi and Ai[512] investigated clauses[xvi] within the conditional expressions contained in 63 Java programs. Figure 7.48 shows the percentage occurrence of conditional expressions containing a given number of clauses; see Github–sourcecode/1-s2.R and Github–sourcecode/sast_2017.R.

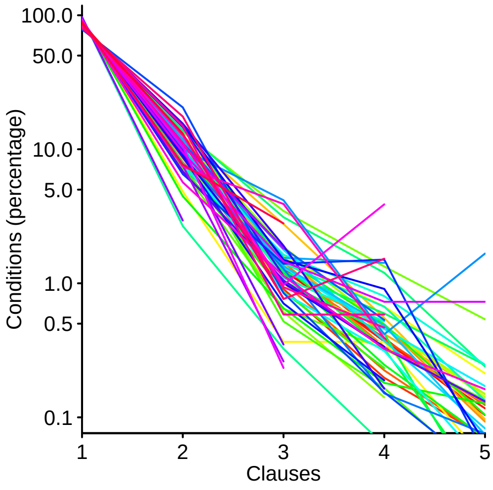


Figure 7.48: Percentage of conditional expressions, in 63 Java programs, containing a given number of clauses; one fitted regression model has the form: $Num\_conditions \propto e^{Num\_predicates \times (\log(SLOC) - 0.6\log(Files) - 11)}$, where each variable is the total for a program's source. Data from Durelli et al.[512] Github–Local

Some languages include statements that provide a restricted form of `goto-statement`, e.g., **break** for jumping just past the end of the associated loop (see fig 11.31). Use of this form removes the need for those reading the code to deduce that the purpose of a **goto**) is to exit a loop (and use of these forms do not have the perceived negative connotations associated with the word **goto**).

Some languages include support for a more powerful form of `goto-statement`, originally based on functionality provided by the hardware, e.g., *signal handling* (known as *exception handling* in some languages). The non-local nature of signal handling (it may cause control flow to exit one or more functions in the call tree) can create a lot of need to know.

A study by de Pádua and Shang[449] investigated exception handling in seven $C^\sharp$ projects (1,502 `try` blocks) and nine Java projects (7,116 `try` blocks). Both $C^\sharp$ and Java support what are known as `try-catch` blocks; if the execution of code within a `try` block raises an exception, it can be caught by the `catch` block (provided the particular exception raised is specified in the list of exceptions handled).

How many exceptions might a `try` block raise? Figure 7.49 shows the number of `try` blocks whose code is capable of raising a given number of exceptions, along with lines showing fitted regression models. Possible reasons for the difference in fitted regression models (i.e., exponential vs. bi-exponential) include: different language characteristics affecting which runtime behaviors are capable of generating an exception, and a consequence of the relatively small number of projects sampled.

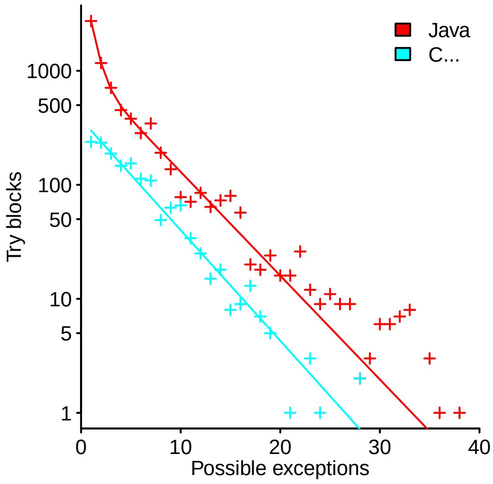


Figure 7.49: Number of `try` blocks whose code might raise a given number of exceptions; fitted regression models have the form: (lower) $Num\_tryBlocks \propto Possible\_exceptions^{-0.22}$ and (upper) $Num\_tryBlocks \propto 7300e^{-1.4 Possible\_exceptions} + 1100e^{-0.21 Possible\_exceptions}$. Data from de Pádua et al.[449] Github–Local

7.3.5 Loops

Loop statements have traditionally been of interest because programs often spend most of their time executing within a few loops (the characteristics of code within loops is intensively studied by compiler writers, optimizing code that commonly occurs in loops is likely to return the greatest ROI for new optimizations).

Compilers attempt to figure out the characteristics of code within loops, such as dependencies between variables in successive iterations, to detect code optimizations[33] that improve the efficiency of the generated code. Calculating worst case program execution time (WCET) requires accurate estimates of the number of iterations, along with the execution time of a single iteration.[1082]

Loops might be classified based on difficulty of automated analysis.[1424]

7.3.6 Expressions

What do developers need to know about the semantics of expression evaluation?

Many languages may perform implicit type conversions on one or more of the operands in an expression, e.g., casting one operand of a binary operator so that both operands have the same type. For instance, many languages consider the operands in the expression `1+1.`

[xvi] A clause is a basic subexpression returning a boolean value, which may be combined with `AND` and `OR` operators to form a more complicated expression.

0 to be some integer type and some floating-point type, respectively, and in languages with C-like implicit conversion rules the behavior is as-if (double)1+1.0 had been written.

The implicit conversions that might be performed vary between languages. For instance, the expression 1+"1" may return the result 2 (e.g., PHP and Lua), or "11" (e.g., Javascript), or generate a compile time error (e.g., many languages), or perhaps something else.

Implicit conversions remove the developer effort needed to write an explicit conversion (and the effort involved in processing its visual form, if the code is later read), but creates a need to know about the implicit conversions specified by the language.

The original need for operands to be converted to a common type, before being operated on, was driven by the behavior of the underlying hardware instructions; the concept of same type was synonymous with same underlying data representation. Some languages have moved away from the concept of types being solely dependent on the underlying representation, and provide a means for developers to specify new type compatibility relationships.

```
type
     celsius is new real;
     fahrenheit is new real;
var
     L_temp  :celsius;
     NY_temp :fahrenheit;
...
 L_temp:=NY_temp; - types not compatible
```

The purpose of developer-defined type constraints is to detect coding mistakes, and their ability to catch mistakes is dependent on the extent to which developers make use of the available functionality. Some languages were explicitly designed to support developer-defined type constraints (e.g., Ada; see margin code), while other language support such functionality through the use of constructs designed for more general uses, e.g., C++ .[211]

When the functionality is available, the extent to which developers make use of user defined type constraints appears to be cultural. For instance, while both Ada and C++ provide mechanisms offering the same level of support for user defined type constraints, there is a culture of developer-defined type constraint use in the Ada community, but not in the C++ community.

The following studies have experimental investigated differences in developer performance when using languages the researchers claim differ in support for strong typing:

- Gannon:[642] used two simple languages, which by today's standards were weakly typed, with one less so than the other (think BCPL and BCPL plus a string type and simple structures). A single problem was solved by subjects, which had been designed to require the use of features available in both languages, e.g., a string oriented problem (final programs were between 50-300 lines). The result data included number of errors during development and number of runs needed to create a working program (this happened in 1977, before the era of personal computers, when batch processing was common; see Github–experiment/Gan77.R).

  There was a small language difference in number of errors/batch submissions; the difference was about half the size of the effect of experimental order of language used by subjects, both of which were small in comparison to the variation due to subject performance differences. While the language effect was small, it was present. It is not possible to separate out performance differences due to stronger typing, rather than built in support for a string type only being available in one language.
- Mayer, Kleinschmager and Hanenberg:[1011, 1208] Two experiments using different languages (Java and Groovy) and multiple problems; the performance metric was time to complete the task. There was no significant difference due to just language, but large differences due to language/problem interaction, with some problems solved more quickly in Java and others more quickly in Groovy, and learning took place (i.e., the second task was completed in less time than the first). As often occurs, there were large variations in performance between subjects (see Github–experiment/mayerA1-oopsla2012.R and Github–experiment/kleinschmagerA1.R).
- Hoppe and Hanenberg:[843] one language (Java) was used, and multiple problems; the problems involved making use of either Java's generic types or non-generic types. Again, the only significant language difference effects occurred through interaction with other variables in the experiment (e.g., the problem or the language ordering), and there were large variations in subject performance.

To summarise: when a language typing/feature effect has been found, its contribution to overall developer performance has been small. Possible reasons for the small or non-existent effect, include: [xvii] the use of subjects with little programming experience (i.e., students; experienced developers are more likely to make full use of the consistency

[xvii] Your author declares his belief that when integrated into the design process, strong typing has cost/benefit advantages.

checking provided by a type system), and the small size of the programs (type checking comes into its own when used to organize, and control, large amounts of code).

Many languages contain more than twenty different kinds of operators which can appear in expressions (supporting the wide variety of different kinds of operations that have been created to combine values). By specifying operator precedence for the relative binding strength of operators (commonly used languages have 10 to 15 precedence levels), to their operands, languages remove the need for developers to explicitly specify the intended binding of operands to operators (by using parenthesis). Expressions that do not use parenthesis create a developer need to know for operator precedence.

One study[920] found that the likelihood of developers knowing the correct relative precedence of two binary operators increased with frequency of occurrence of the respective pair of operators in existing C source; see fig 2.38.

7.3.6.1 Literal values

Literal values appear in source for a variety of reasons, including: specific value required by an algorithm,[71] size or number of elements in the definition of an array,[919] implementation specific values (e.g., urls, dates and developer credentials[2004]), application domain values, personal preferences of developers (see section 2.7.1), and a representation of no-value (i.e., a null value).

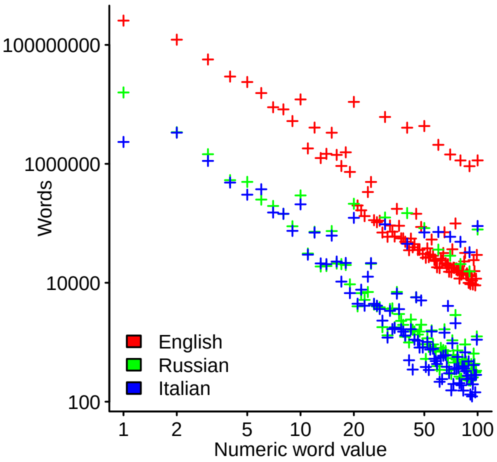


Figure 7.50: Yearly occurrence of number words (e.g., "one", "twenty-two"), averaged over each year since 1960, in Google's book data for three languages. Data kindly provided by Piantadosi.[1464] Github–Local

The distribution of numeric values in application domains will have been influenced by real-world usage. Figure 7.50 shows the yearly occurrence of number words (averaged over each year since 1960) in Google's book data. The English counts are larger because most of the books processed were written in English. Decade values (e.g., ten, twenty) follow their own trend, and these are much more common than adjacent values.

The use of the value zero during the execution of many kinds of program is sufficiently common that many RISC processors hard-code one register to contain zero; the use of the whitespace character is very common in Cobol applications.[xviii]

Some languages support multiple ways of representing numeric literals (e.g., decimal, binary, hexadecimal). Figure 13.5 suggests that the distribution of the value of numeric literals depends on the representation used. Figure 7.51 shows that Benford's law is a very crude approximation for decimal integer and floating-point numeric literal usage in source code.

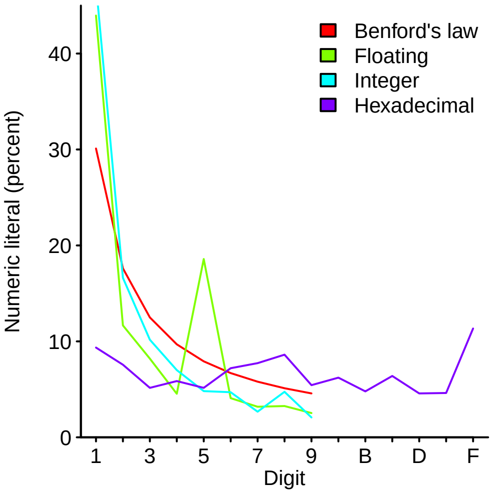


Figure 7.51: Percentage occurrence of the most significant digit of floating-point, integer and hexadecimal literals in C source code. Data from Jones.[919] Github–Local

7.3.6.2 Use of variables

What are the patterns of use of variables in source code?

Section 8.3.1 discusses models that relate frequency of local variable use and number of variable declarations, within in a function.

A study by Sajaniemi and Prieto[1607] investigated the roles of variables in source code. They were able to categorise variable use into one of approximately 10 roles, which included: *stepper* which systematically takes predictable successive values, *follower* which obtains its new value from the old value of another variable, and *temporary* which holds some value for a short time.

Variable use may be driven by the constructs supported by the language. For instance, in C, the loop header often contains three appearances of the loop control variable, e.g., `for (i=0;i<10;i++)`; in languages that support constructs of the form `for (i in v_list)`, only one appearance is required. In languages that support vector operations, an explicit loop may not be needed to perform some operations on variables, e.g., in R two vectors can be added together using the binary plus operator.

An analysis[373] of integer use in C found that around 20% of accesses to variables, having an integer type, were made in a context having a signedness that was different from the declared type; also the declarations of variables having an integer type were not usually modified.

How often are variables read and written by functions? One study[919] measured C source, with variables local to the function, source file or externally visible. Figure 7.52 shows:

[xviii] The MicroFocus Cobol code generator for the SPARC processor, designed by your author, dedicated one 32-bit register to always hold the value 0x20202020, i.e., four whitespace characters.

upper the number of functions containing a given number of references to the same variable (the same function may be counted more than once), and lower: the number of functions containing a given number of references to all variables; solid lines are reads, dashed lines are writes. Most functions reference a few variables, which is consistent with most functions containing a few lines (see fig 7.14).

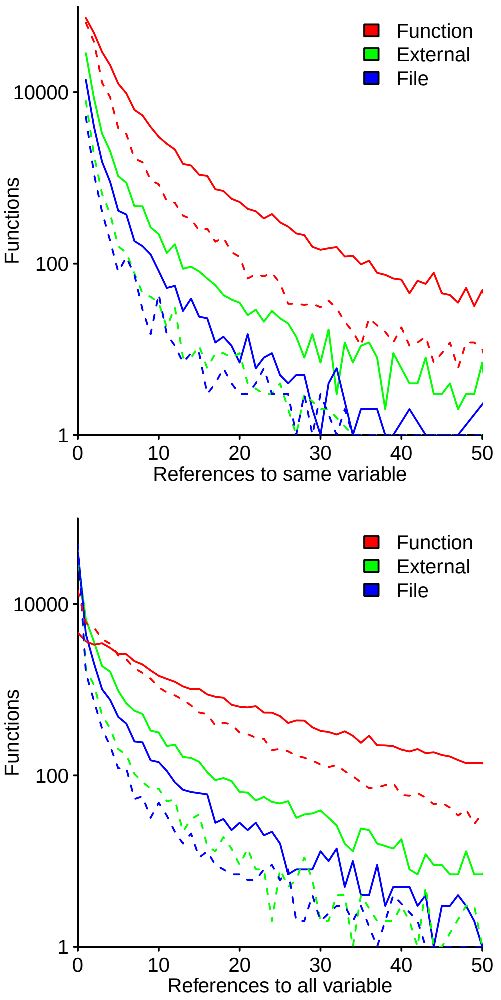


Figure 7.52: Number of C functions contains a given number of references to the same variable (upper), and a given number of references to all variables (lower); reads are full lines, writes dashed lines, colors indicate variable's visibility. Data from Jones.[919] Github–Local

A study by Gonzaga[694] investigated the use of global variables and parameters in the functions defined in 40 C programs. Comparing the number of function parameters, functions that did not access global variables had 0.4 more parameters (on average). For 30 programs the larger number of parameters, for functions accessing/not accessing global variables, was statistically significant (see Github–sourcecode/Gonzaga.R).

Figure 7.53 shows the number of functions defined to have a given number of parameters; solid lines are functions that did not access global variables, dashed lines are functions that accessed global variables.

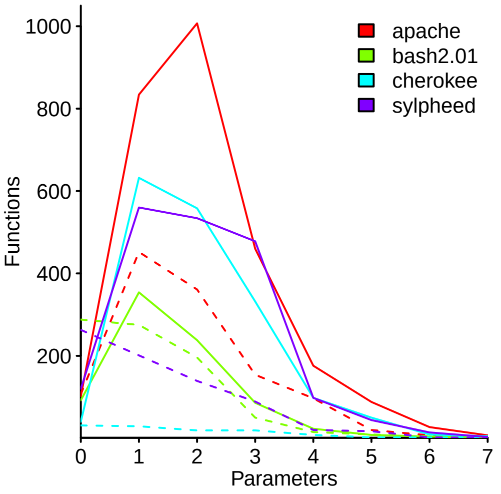


Figure 7.53: Number of functions defined with a given number of parameters in the C source of four projects; solid lines function body did not access global variables, dashed lines function body accessed global variables. Data from Gonzaga.[694] Github–Local

7.3.6.3 Calls

Roughly 1-in-5 statements contains an explicit function/method call (compilers sometimes introduce additional calls to implement language constructs; see fig 7.5).

Some call sequences are part of a common narrative, e.g., open a file, write to it and close it. Detecting narrative sequences can be straightforward in code written in object-oriented language, because the variable name associated with an object is included in calls to methods associated with that object, e.g., `var.strLength()`.

A study by Mendez, Baudry and Monperrus[1246] investigated method call sequences associated with the same variable, based on an analysis of 4,888 classes in 3,418 Jar files (i.e., Java bytecode; some method calls may not explicitly appear in the original source, e.g., the compiler maps string concatenation using binary + to a call to the method `StringBuilder.append`). Figure 7.54 shows the 10 most frequent sequences of `java.lang.StringBuilder` methods called on the same variable (lines connect methods called in sequence; method call argument types are ignored).

Figure 7.55 shows the number of sequences having a given length (i.e., measured in methods; in blue), and number of sequences that appear in the code (i.e., are used; in red) a given number of times; only classes containing method sequences used at least 100 times are included.

How does the number of calls to distinct methods grow with project size?

A study by Lämmel, Pek and Starek[1061] investigated calls to methods from third-party APIs, and those defined within 1,435 projects. Figure 7.56 shows the number of distinct API methods called against project size (measured in method calls).

The function/method called may be passed as an argument (i.e., a *callback*). A study by Gallaba, Mesbah and Beschastnikh[637] investigated the use of callbacks in 130 Javascript programs. Figure 7.57 shows the total number of calls in each program, against the number of calls containing callbacks, and just anonymous callbacks.

7.3.7 Declarations

Declarations[xix] are code bureaucracy that provides two basic services: a means of introducing a sequence of characters that is to be treated as a valid identifier (sometimes known as a *name*), and specifying operations associated with uses of the identifier in source code (these operations are derived from information appearing in the declaration of the name, e.g., the type of an object).

Large programs may define tens of thousands of identifiers.[919]

Some languages do not require an identifier to be defined in a declaration, before (or after) it appears in the source code. In such languages the context in which the identifier appears is used to derived attributes associated with subsequent uses, e.g., variables have the type of the value last assigned to them. A study[550] of four large PHP applications found that less than 1% of variables were assigned values having different types (e.g., assigning an array and later assigning an integer).

[xix] Some languages use the term *definition*, and some use both, e.g., in C a definition is a declaration that causes storage to be allocated for an object.

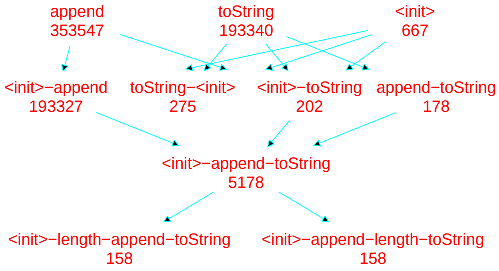


Figure 7.54: Sequences of methods, from `java.lang.StringBuilder`, called on the same object; based on 3,418 Jar files. Data from Mendez et al.[1246] Github–Local

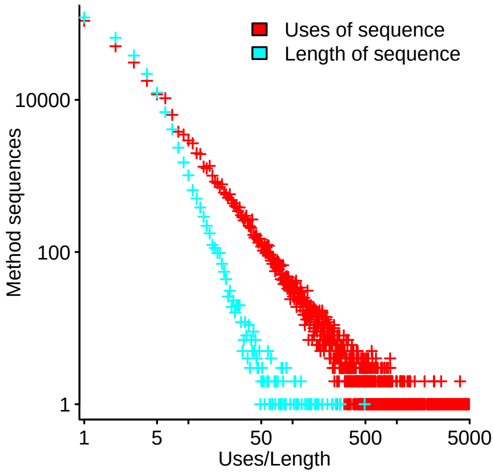


Figure 7.55: For each Java class, in 3,418 jar files, the number of method sequences containing a given number of calls (red), and the number of uses of each sequence (blue). Data from Mendez et al.[1246] Github–Local

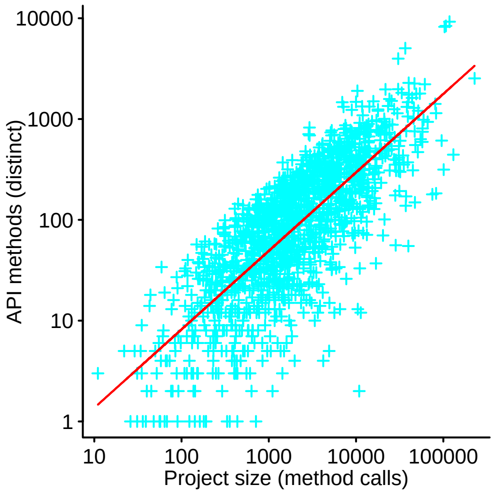


Figure 7.56: Number of distinct API methods called in 1,435 Java projects containing a given number of method calls; the line is a fitted regression model of the form: $unique \propto calls^{0.78}$. Data from Lämmel et al.[1061] Github–Local

Desirable characteristics of declarations are those that minimise the need to know about the identifiers defined.

Some degree of visibility is one characteristic identifiers acquire during the definition process (i.e., they can be referred to over some region of the source code). Reducing the visibility of identifiers reduces the amount of information developers need to know when dealing with code where the identifiers are not required to be visible.

Many languages provide mechanisms for restricting the visibility of identifiers (e.g., the `private` in Java and `static` in C; `R` is an example of a language that provides limited functionality). Studies[1881] have found that identifiers are sometimes declared with greater visibility than necessary; see Github–sourcecode/TR_DCC-overExposure/TR_DCC-overExposure.R.

To what extent do declarations change over time?

A study by Neamtiu, Foster and Hicks[1340] investigated the release history of three C programs over 3-4 years, and a total of 48 releases. They found that one or more fields were added to one or more existing structure or union types in 79% of releases, while structure or union types had one or more fields deleted in 51% of releases; a later study[1341] found one or more existing fields had their types changed in 35% of releases. Figure 7.58 shows the relationship between the number of global variables and lines of code, in three C programs, over multiple releases.

A study by Robbes, Róthlisberger and Tanter[1568] investigated data extensions (i.e., the visitor pattern) and operation extensions to Smalltalk classes; the 2,505 projects analyzed contained 95,662 classes, forming 48,595 class hierarchies (with 41% containing more than one class). Figure 7.59 shows the number of data and operation extensions made to 1,560 class hierarchies containing both kinds of extension.

One study[1335] of eight Java systems found that the number of methods and classes having a given inheritance depth decreased by a factor of 0.25 per inheritance level; see Github–sourcecode/JavaInherit.R

7.3.8 Unused identifiers

Some identifiers are defined, and never referenced again, i.e., they are unused. Reasons for the lack of use include: a mistake has been made (e.g., the variable should have been referenced), the identifier was once referenced (i.e., the declaration is now redundant), and there is an expectation of future need for the entity that has been defined.

Unused identifiers consume cognitive resources for no benefit.

It is not always cost effective to remove the definition of an unused identifier. For instance, removing an unused function parameter may require a greater investment than is likely to be worthwhile (because changing the number of function parameters requires corresponding changes to the arguments of all calls).

Figure 7.60 shows the total number of functions having a given number of parameters, a given number of unused parameters, and various fitted regression models. Unused function parameters, which at around 11% of all parameters are slightly more common than unused local variables.

The fitted regression model, for the number of functions containing a given number of unused parameters has the form: $functions \propto e^{-0.5unused}$ (in practice, functions are likely to acquire unused parameters one at a time, as the code evolves). An alternative formula for estimating the number of functions containing u unused parameters is: $\sum_{p=u}^{8} \frac{F_p}{7p}$, where: F_p is the total number of function definitions containing p parameters; figure 7.60 shows how well this wet-finger model fits.

7.3.9 Ordering of definitions within aggregate types

Consistent patterns appear in the ordering of member declarations within many Java `class` and C `struct` types; for instance, members sharing an attribute are often sequentially grouped together, or have a preferred relative ordering.

These usage patterns may be the result of many developers making individual choices (either explicitly or implicitly), or because externally specified ordering rules are being followed, e.g., the Java coding conventions (JCC)[1774] specifies a recommended ordering for the declaration of: class variables (or fields), instance variables (or static initializers), constructors and methods (see fig 12.8).

Statistical analysis techniques for items believed to have a preferred order are discussed in section 12.4.

Patterns in the ordering of fields in C `struct` types are discussed in section 9.6.1. One interpretation of the pattern seen, is developers wanting to reduce unused storage by grouping together objects whose types have the same alignment requirements.

A study by Geffen and Maoz[656] investigated various patterns of method ordering within Java classes; for instance, the rules specified by the `StyleCop` tool (e.g., group by access modifiers), the commonly seen pattern of called methods appearing after the method that involved them, and the concept of clustering (i.e., related methods appearing in the same class or file).

A study by Biegel, Beck, Hornig and Diehl[193] investigated the impact of the kind of activity performed by a method on its relative ordering. The method activity attributes considered were: 1) all static methods (declared using the `static` keyword), 2) initializers (method name begins with `init`), 3) getters and setters (non-void return type and method name begins with `get`, `is` or `set` followed by a capital letter) and 4) all other non-static methods.

Figure 7.61 shows that methods performing two of the activities are very likely to be ordered before methods performing the two remaining other activities. Method declaration sequences containing more than two kinds of method activity occurred in 62% of contexts analysed, with 54% of these passing the threshold needed for analysis, leaving 33% of all declarations sequences.

The original author of the code may not be responsible for maintenance, and it is possible that declarations added during maintenance were not inserted into an existing structure/class declaration according to the ordering pattern used during initial development, e.g., some developers may prefer to add new declarations at the end of an existing structure/class.

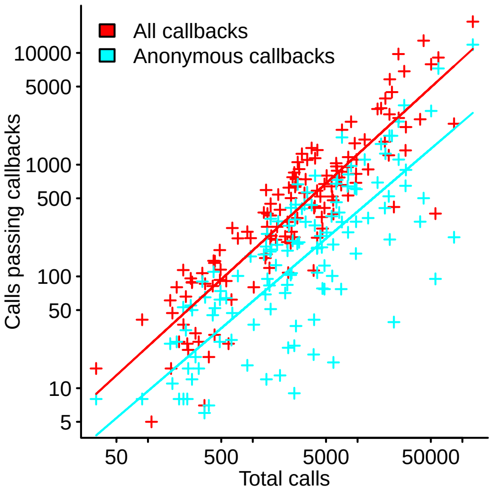


Figure 7.57: Number of function calls, against corresponding number of calls containing callbacks and anonymous callbacks, in 130 Javascript programs; lines are fitted regression models of the form: $allCallbacks \propto allCalls^{0.86}$ and $anonCallbacks \propto allCalls^{0.8}$, respectively. Data from Gallaba et al.[637] Github–Local

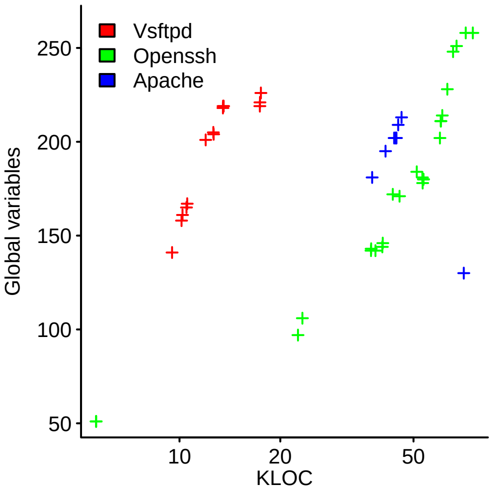


Figure 7.58: Number of global variables against lines of code over 48 releases of three systems written in C. Data kindly provided by Neamtiu.[1340] Github–Local

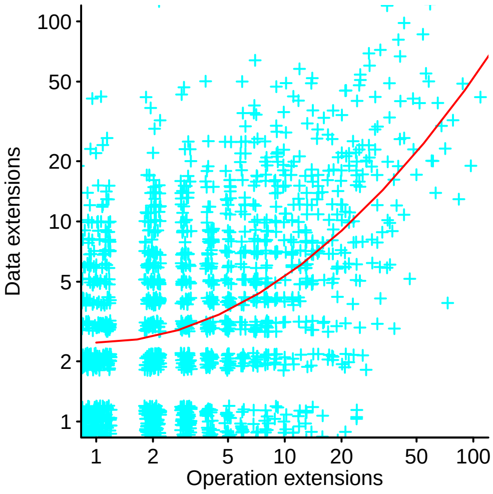


Figure 7.59: Jittered number of data and operation extensions to 1,560 Smalltalk class hierarchies containing both kinds of extension; regression line has the form: $\log(Data\_extensions) \propto \log(Operation\_extensions)^2$. Data from Robbes et al.[1568] Github–Local

7.4 Evolution of source code

Software only changes when developers have an incentive to spend time making the changes, incentives include: being paid, and a desire to change the code to satisfy a personal need, e.g., refactoring code to maintain a personal self-image, as a developer, because of a belief, for instance, that other developers reading the unrefactored code would form a low opinion of the author.

If payment is involved, there is a customer, and the changes are supposed to address customer needs (it can be very difficult to work out what the customer needs actually are, and there may be as many opinions about these needs as there are people trying to keep the customer happy).

The uncertainty of future customer demand creates uncertainty in the cost/benefit analysis of investment decisions.

Software systems growth, in lines of code, over time is a commonly quoted metric; reasons for growth include improvements to existing functionality and the addition of new functionality. Some systems grow at a consistent rate over many years (e.g., for FreeBSD see fig 11.2, and for the Linux kernel see fig 11.7), while others appear to have stopped adding lines (e.g., the glibc library, see fig 11.52), or grow sporadically (e.g., the Groovy compiler, see fig 11.9).

Growth can increase interdependencies between components; figure 7.62 shows the relationship between the separate components of ANTLR over various releases.

Factors influencing the rate of evolution of source code characteristics include:

- customer limited: insufficient customer demand (as measured by willingness to pay) for it to be economically worthwhile updating existing functionality (to support changes in the world), or adding new functionality, e.g., new hardware requiring device drivers,

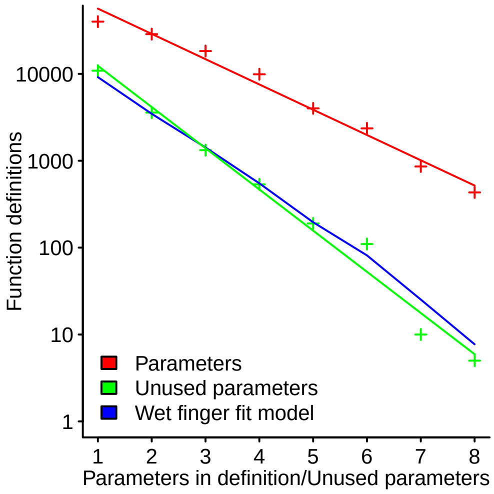


Figure 7.60: Number of C function definitions having a given number of parameters (red) and unused parameters (green); parameter fitted regression line has the form: $functions \propto e^{-0.67 parameters}$. Data from Jones.[919] Github–Local

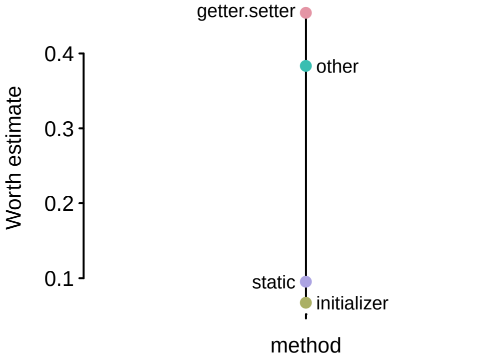


Figure 7.61: "Worth estimate" for the kind of method activity attribute (see section 12.4). Data from Biegel et al.[193] Github–Local

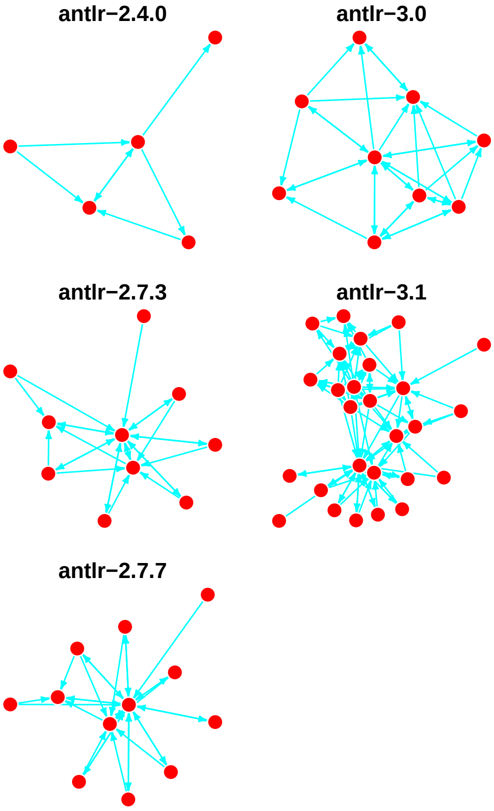


Figure 7.62: Dependencies between the Java packages in various versions of ANTLR. Data from Al-Mutawa.[28] Github–Local

- developer limited: bottlenecks in the development process that restrict the quantity of change per unit time. For instance, a limited number of people with the necessary skills, change requests requiring sign-off by a handful of senior managers, or increasing developer resources required to support a growing system leading to diminishing returns from adding more developers,
- competition from other applications: source code may cease to evolve because its host, the application, is out-competed, e.g., customers stop using the application and/or it looses developer mindshare,
- hardware characteristics: there may be benefits to adapting software to the characteristics of the hardware on which it is used.

  In Fortran, **common** blocks provide a means of specifying how different variables are overlaid in memory; for several decades **common** blocks were widely used. As the amount of memory available on computers grew, and compilers became more sophisticated at optimizing memory allocation, the need to use **common** decreased.[xx]

A consistent rate of code growth suggests some degree of consistency in demand for new updates, and developer resources available to do the work; see fig 11.2.

During the evolution of source code some of the contents of units of code (e.g., files or functions) may be moved to other units.[682] Studies of code evolution that do not take code migration into account will overestimate the amount of code added and deleted, over time.

Updating existing functionality may result in source code being deleted.

Figure 7.63 shows the percentage of code in 130 releases of Linux that originated in earlier releases, and fig 4.18 shows code shared between different releases of related BSD operating systems; fig 11.70 shows the correlation between lines added/deleted for glibc, fig 9.21 shows a Markov chain for the creation/modification/deletion of files in the Linux kernel.

7.4.1 Function/method modification

The likelihood of modifying existing code is an essential input to the cost/benefit analysis carried out prior to making any investment intended to reduce the cost of future modifications. The expected lifespan of the system containing the code is a higher level consideration discussed in section 4.2.2.

A new function/method definition is about to be written, and it is believed that at some future time it may need to be modified. If an investment, I, in extra work is made today to receive the benefit, B, for each of the M_t future modifications: like all investments, the expected benefit is required to be greater than the investment, e.g., $I < M_t B$.

Let s be the likelihood that a function is modified in the future, and that once modified the likelihood of it being modified again remains unchanged; the expected number of modifications of a given function is then: $M_t = s + 2s^2 + 3s^3 + \cdots + ns^n$, where: n is the maximum number of modifications of a function; this series sums to:

$$M_t = \frac{s - (n+1)s^{n+1} + ns^{n+2}}{(1-s)^2}$$

substituting and rearranging the cost/benefit equation, and assuming $(n+1)s^{n+1}$ is very small, gives:

$$\frac{(1-s)^2}{s} < \frac{B}{I}$$

What range of values might s have in practice? A study by Robles, Herraiz, German and Izquierdo-Cortázar[1573] analysed the change history of functions in Evolution (114,485 changes to functions over 10 years), and Apache (14,072 changes over 12 years).

Figure 7.64 shows the number of functions (in Evolution) that have been modified a given number of times (upper), and the number of functions modified by a given number of different authors (lower). A bi-exponential model provides a reasonable fit to both sets of data. One interpretation of this bi-exponential model is that many functions are modified by the same developer (or core team members) during initial implementation (see

[xx] A common Fortran coding mistake was to assign to a variable sharing the same memory location as another variable, and later to access the other variable believing it contained what was earlier assigned to it, i.e., the lifetimes of variables stored in the same memory location overlapped.

figure 7.66), with fewer functions modified after initial development (with non-core developers more likely to be involved).

The previous analysis assumes s is constant (i.e., the data is fitted by one exponential), but figure 7.64 is fitted using a bi-exponential (which has a non-constant s). The mean half-life of the bi-exponential: $ae^{-\lambda_1 x} + be^{-\lambda_2 x}$, is: $\tau_{mean} = \dfrac{a\tau_1^2 + b\tau_2^2}{a\tau_1 + b\tau_2}$, where: $\tau_1 = \frac{1}{\lambda_1}$ and $\tau_2 = \frac{1}{\lambda_2}$.

Using τ_{mean} gives, for Evolution: $s = 0.64$, and $0.56 < \frac{B}{I}$. Using the post initial development exponential, $s = 0.85$, and $47 \times 0.025 = 1.2 < \frac{B}{I}$ (the original investment was made in 47 times as many functions/methods as fitted by this exponential; see Github–evolution/author-mod-func.R for details).

For Apache, the mean τ_{mean} gives: $s = 0.81$, and $0.046 < \frac{B}{I}$, and post initial development is: $s = 0.95$, and $96 \times 0.0032 = 0.3 < \frac{B}{I}$.

This model does not take into account any benefits received if developers read the code without modifying it.

Figure 7.65 shows the number of modifications of a function, stratified by number of authors. The form of the following equation was found by trial and error, it fits the data reasonably well: $\log(num\_authors)^{0.2}(\alpha + \beta num\_mods^{0.3}) + \gamma num\_mods^{0.3}$, where: α, β and γ are fitted constants.

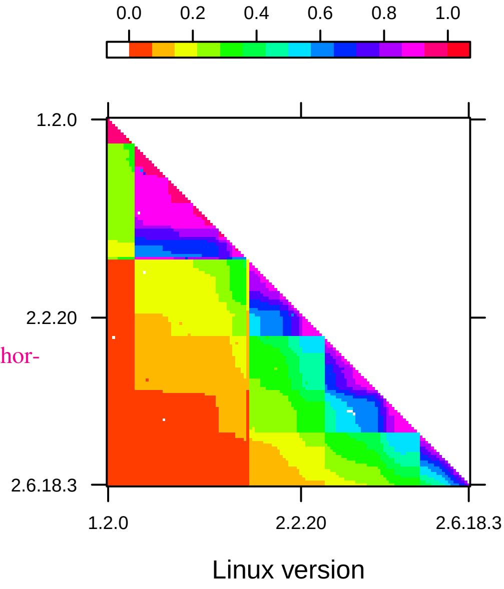


Figure 7.63: Fraction of source in 130 releases of Linux (x-axis) that originates in an earlier release (y-axis). Data extracted from png file kindly supplied by Matsushita.[1143] Github–Local

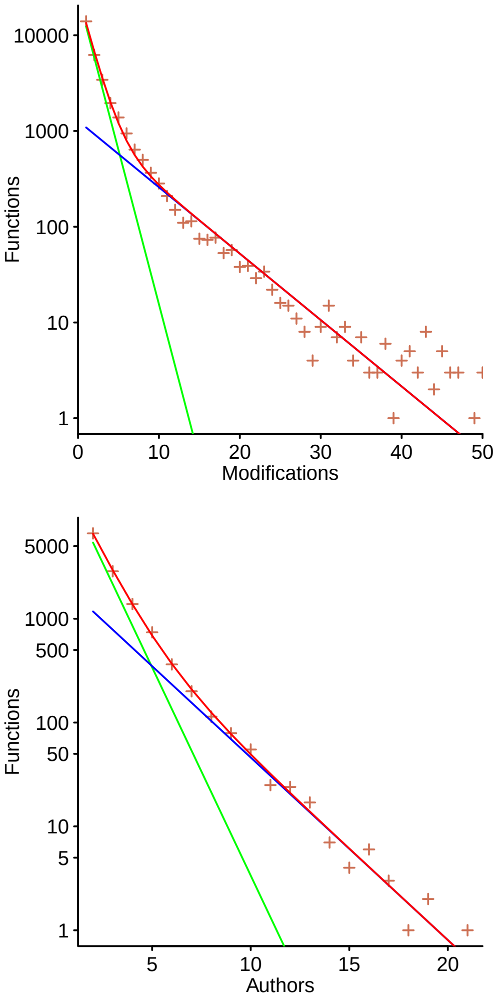


Figure 7.64: Number of functions in Evolution modified a given number of times (upper), and modified by a given number of different people (lower); red line is a fitted bi-exponential, green/blue lines are the individual exponentials. Data from Robles et al.[1573] Github–Local

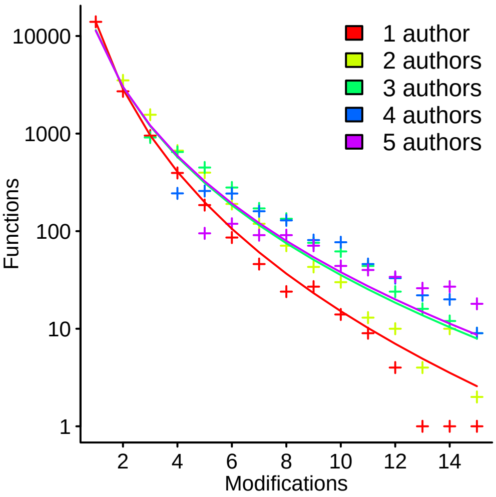


Figure 7.65: Number of functions (in Evolution) modified a given number of times, broken down by number of authors; lines are a fitted regression model. Data from Robles et al.[1573] Github–Local

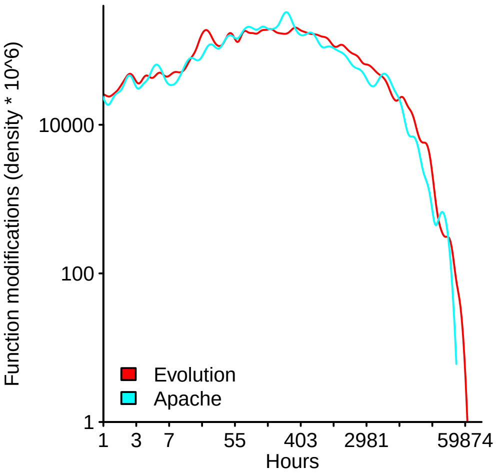


Figure 7.66: Density plot of the time interval, in hours, between each modification of the functions in Evolution and Apache. Data from Robles et al.[1573] Github–Local

Chapter 8

Stories told by data

8.1 Introduction

Data analysis is the process of finding patterns in data and weaving a story around these patterns.

Finding patterns in data is easy, weaving a believable narrative around them can be very difficult. Figure 8.1 may be interpreted as evidence for a causal connection between UFO activity and computer virus infections. Domain knowledge (e.g., personal experience of reporting problems and events) might lead us to believe that these reports were made by people, and an alternative interpretation is that U.S. counties with larger populations experienced and reported more virus infections and UFO sightings, compared to counties having smaller populations.

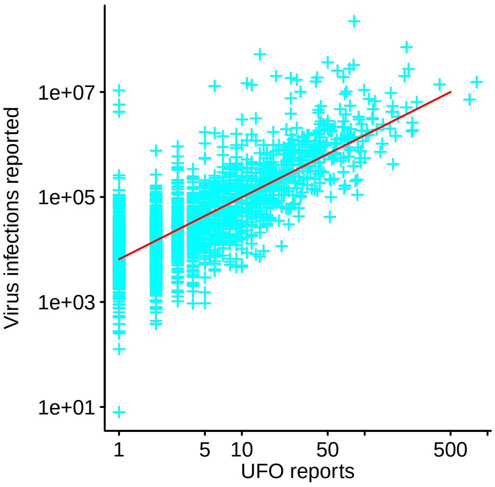


Figure 8.1: Number of virus infections and UFO sighting, reported in 3,072 U.S. counties during 2010; the line is a fitted regression model having the form: $virus\_reports \propto UFO\_reports^{1.2}$. Data from Jacobs et al.[897] Github–Local

An understanding of common patterns found in data is the starting point for an appreciation of the kinds of stories that these patterns might be used to substantiate. This chapter starts with an overview of techniques that may be used to uncover patterns in data, before moving on to discussing the communication of these patterns to others. Those performing the analysis are responsible for weaving a story around the patterns found; the figures, and numeric values provide props that may be used to conjure a convincing narrative.

The patterns sought have the form of a relationship between two or more measured quantities. Managers want to control software development, and to do this they need understanding of the processes that are driving it. Regression modeling is this book's default technique for modeling the relationships between the quantities that have been measured; see chapter 11.

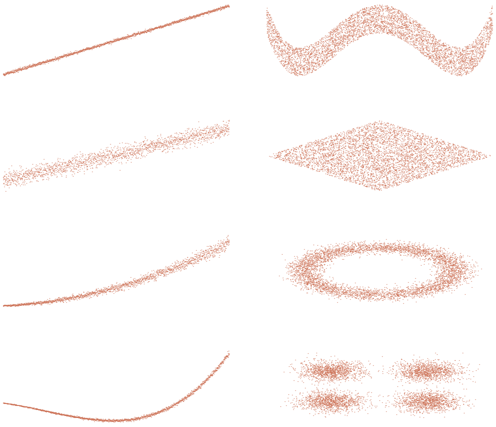


Figure 8.2: Data having values following various visual patterns, when plotted. Github–Local

Ideally you, the data analyst, have:

- sufficient domain knowledge to be able to distinguish between spurious correlations that may be present in the data, and correlations connected to the processes that generated the data,
- practical ideas relating to the questions for which answers are sought, in practice there may be a lot of uncertainty about what the questions are.

  Questions have to have answers that can be used to make predictions about expected patterns of behavior in the data (which can be searched for).

  If a question does not have an associated answer that has a predictable, detectable, pattern of behavior, then 42 is as good an answer as any other,
- the time and resources needed to obtain data likely to contain answers to the questions asked; obtaining data is often time-consuming and/or expensive and it is often necessary to make do with whatever data is cheaply and quickly available (even if it only indirectly relate to the questions being asked). This book generally assumes that a dataset has been obtained, some of the issues around obtaining data are discussed in chapter 13.

  The data should contain as little noise, in practice the available data may be very noisey and cleaning may be very time-consuming,
- the ability to deal effectively with uncertainty, and an awareness of personal cognitive biases,[814]

- statistical analysis techniques capable of providing answers to the desired level of certainty; in practice it may not be possible to draw any meaningful conclusions from the data or more questions will be uncovered.

Data analysis is like programming, in that people get better with practice; there are a few basic techniques that can be used to solve many problems and doing what you did on a previous successful project can save lots of time.

This, and subsequent chapters explicitly discuss the R code that was used (previous chapters discuss the results of data analysis, not how the analysis was done).

There is no guarantee that the available data contains any information that might be used to answer any of the questions being asked of it.

Considerations used to evaluate possible interpretations of patterns found in data include: model simplicity, consistency with existing models of how things are believed to work, and how well a model fits the available data. If the data does not contain population information (and it cannot be easily obtained), the extent to which this alternative interpretation is consistent with the report data can be checked. Without appropriate data, alternative interpretation is based on the analysts model of the world from which the data was obtained.

At a bare minimum, the story told by an analysis of data needs to meet the guidelines for truthfulness in advertising that is specified by the national advertising standards' authority. If manufacturers of soap powder have to meet these requirements, when communicating with the public, then so should you.

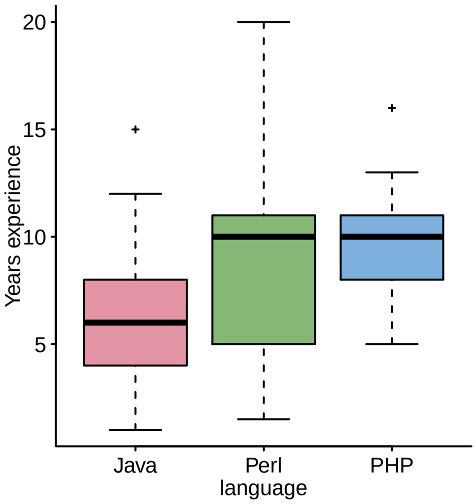


Figure 8.3: Years of professional experience in a given language for experimental subjects. Data from Prechelt.[1503] Github–Local

Check assumptions derived from visualizations Assumptions suggested by a visualization of data need to be checked statistically. For instance, Figure 8.3 shows professional software development experience, in years, of subjects taking part in an experiment using a particular language. The visual appearance suggests that as a group, the PHP subjects are more experienced than the Java subjects. However, a permutation test, comparing years of experience for the PHP and Java developers, shows that the difference in mean values is not significant (there are only nine subjects in each group, and the variation in experience is within the bounds of chance; see Github–communicating/postmortem-answers.R).

8.2 Finding patterns in data

When a specific pattern is expected, the data can be checked to see whether it contains this pattern. Otherwise, the search for patterns is essentially a fishing expedition.

Figure 8.2 shows some common and less common patterns seen in data. The left column shows data forming lines of various shapes; a straight line is perhaps the most commonly encountered pattern in data and points may all be close to the line or form a band of varying width. The right column shows data clustering together in various regular shapes. Uncovering a pattern is the next step along the path to understanding the processes that generated the sample measurements.

Vision is the primary pattern detection pathway used in this book. Animals have developed sophisticated visual pattern detection and recognition systems (see section 2.3); number processing is a very new ability, and as such is relatively slow and unsophisticated.

Compelling numbers. For small quantities of numeric data, the pattern present in the printed form of the values may be the most compelling visual representation. For instance, relative spacing is sometimes used within the visible form of expressions to highlight the relative precedence of binary operators (e.g., more whitespace around the addition operator when it appears adjacent to a multiplication, e.g., `5 + 2*3`). Table 8.1 shows that when relative spacing is used, it nearly always occurs in a form that where the operator with higher precedence has closer proximity to its operands (relative to the operator having a lower precedence). The number of cases where the reverse occurs is small, suggesting that either the developer who wrote the code did not know the correct relative precedence or there is a fault in the code.

A study by Landy and Goldstone[1068] found that subjects were more likely to give the correct answer (and answer more quickly) to simple arithmetic expressions, containing two binary operators, when there was greater visual proximity between the operands that were separated by the binary operator having the higher precedence.

	Total	High-Low	Same	Low-High
no-space	34,866	2,923	29,579	2,364
space no-space	4,132	90	393	3,649
space space	31,375	11,480	11,162	8,733
no-space space	2,659	2,136	405	118
total	73,032	16,629	41,539	14,864

Table 8.1: Number of expressions containing two binary operators having the specified spacing in the visible source (i.e., no spacing, no-space, or one or more whitespace characters {excluding newline}, space) between a binary operator and both of its operands. The High-Low column lists counts for expressions where the first operator of the pair has the higher precedence (some are expressions where the both operators of the pair have the same precedence), the Low-High column lists counts for expressions where the first operator of the pair has the lower precedence. For instance, `x + y*z` is space no-space because there are one or more space characters either side of the addition operator and no-space either side of the multiplication operator, the precedence order is Low-High. Data from Jones.[919]

8.2.1 Initial data exploration

Initial data exploration starts with the messy issue of how the data is formatted (lines containing a fixed number of delimited values is the ideal form, because many tools accept this as input; if a database is provided it may be worth extracting the required data into this form).

A programmer's text editor is as good a tool as any for an initial look at data, unless the filename suggests it is a known binary format (e.g., spreadsheet or database). For data held in spreadsheets exporting the required values to a csv file is often the simplest solution.

This initial look at the data will reveal some basic characteristics, such as: number of measurement points (often the number of lines) and number of attributes measured (often the number of columns), along with the kind of attributes recorded (e.g., date, time, lines of code, language, cost estimated, email addresses, etc).

The most important reason for viewing the file with an editor, first, is to identify the character used to delimit columns.

A call to `read.csv` reads the entire contents of a text file into a data frame (what R calls a structure or record type). The file is assumed to contain rows of delimited values (there is an option to change the default delimiter); spurious characters or missing column entries can cause subsequent values to appear in the incorrect column (chapter 14 provides some suggestions for finding and correcting problems such as this). The `foreign` package contains functions for reading data stored in a variety of proprietary binary forms.

Having read the file into a variable, the following functions are useful for forming an initial opinion of the characteristics of the data that has been read (unless the dataset is small enough to be displayed on a screen in its entirety):

- `str` returns information about its argument, e.g., the number of rows and columns, along with the names, types and first few values of each column in a data frame,
- `head` and `tail` print six rows from the start/end of their argument respectively,
- `table` prints a count of the number of occurrences of each value in its argument, e.g., a particular column of a `data.frame` (by default `NA`s are not included). The `cut` function can be used to divide the range of its argument into intervals, and return the bounds of the intervals and the corresponding counts in each interval.

If `str` reports a column having an unexpected type (e.g., `chr` rather than `int`), the likely causes are missing values and spurious characters in the data.

When one or two columns are of specific interest, `plot` can be used to quickly visualize the specific values of interest. Chapter 14 discusses techniques for cleaning data.

Figure 8.4, upper plot, shows a very noticeable change in the number of occurrences, in C source files, of lines having a given length (i.e., number of characters on a line). What might cause this pattern to occur?

The change occurs at around the maximum line length commonly supported by non-GUI, non-flat screen, terminals (these measurements are of C source that is over 10 years old, i.e., before flat screen monitors became available). One hypothesis is that a system limit has a significant impact on the usage characteristics. A prediction derived from this hypothesis is that code written by developers using terminals that supported more characters per line would contain a greater number of longer lines, i.e., the downturn in the plot would move to the right.

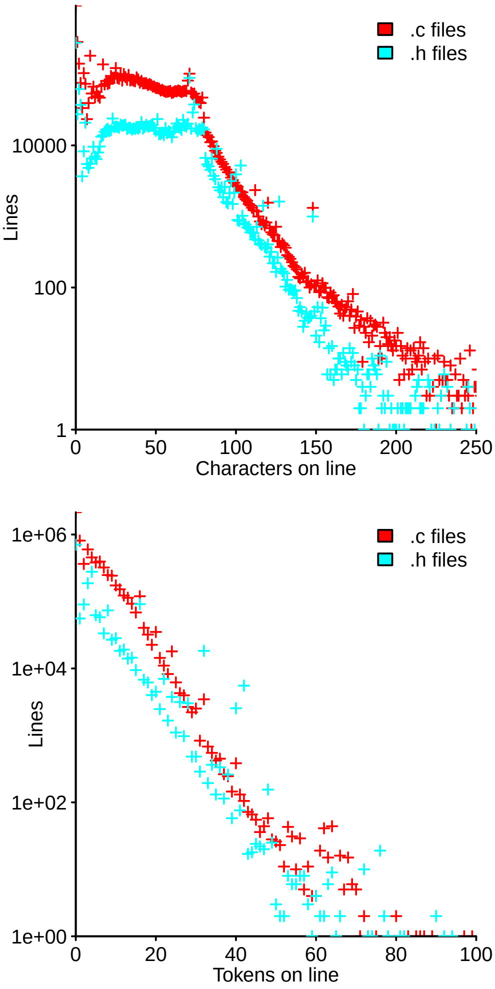


Figure 8.4: Total number of lines of C source, in .c and .h files, having a given length, i.e., containing a given number of characters (upper) and tokens (lower). Data from Jones.[919] Github–Local

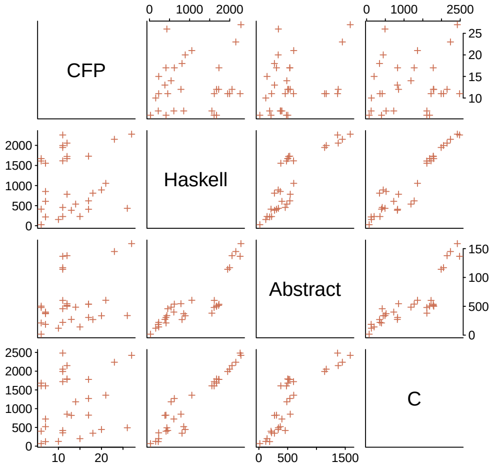


Figure 8.5: Various measurements of work performed implementing the same functionality, number of lines of Haskell and C implementing functionality, CFP (COSMIC function points; based on user manual) and length of formal specification. Data kindly provided by Staples.[1740] Github–Local

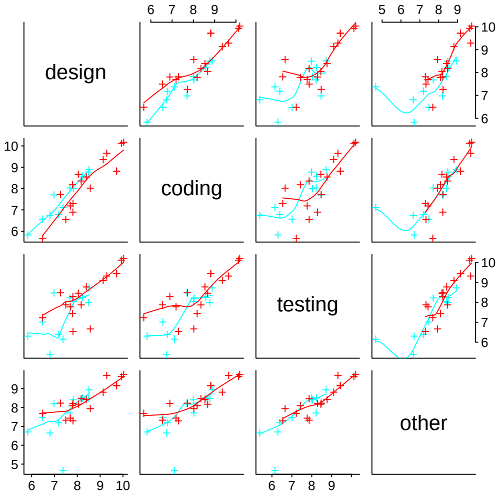


Figure 8.6: Effort, in hours (log scale), spent in various development phases of projects written in Ada (blue) and Fortran (red). Data from Waligora et al.[1897] Github–Local

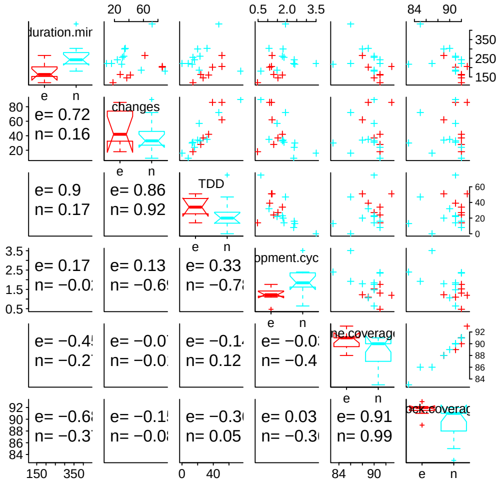


Figure 8.7: Performance of experts (e) and novices (n) in a test driven development experiment. Data from Muller et al.[1308] Github–Local

Figure 8.4, lower plot, illustrates that a different representation of the same information may not have any immediately obvious visual pattern. This plot is a count of the number of tokens per line. Knowing that average token length is around 3-4 characters, suggests that the slight change in the downward slope of the data points just visible at around 25 tokens corresponds to the more dramatic dip seen in the characters-per-line plot.

When a data set contains many variables, plotting one pair of variables at a time is an inefficient use of time. The `plot`, when given a data frame containing three or more columns, creates nested plots of every pair of columns. Figure 8.5 shows four sets of measurements relating to the same task; some measurement pairs are in a roughly linear relationship, while no obvious visual pattern is apparent for other pairs.

```
work=read.csv(paste0(ESEUR_dir, "communicating/pub-fs-fp.csv.xz"), as.is=TRUE)
# -1 removes the first column
plot(work[, -1], col=point_col, cex.labels=2.0)
```

A list of columns can be specified using the formula notation; the following code has the same effect as the previous example:

```
plot( ~ CFP+Haskell+Abstract+C, data=work[, -1], col=point_col, cex.labels=2.0)
```

If a more tailored visualization of pairs of columns is required, the `pairs` function supports a variety of options. For instance, separating out and highlighting subsets of a sample (known as *stratifying*) can be used to highlight differences and similarities. Figure 8.6 separates out measurements of Ada and Fortran projects. The lines are from fitting the points using loess, a regression modeling technique (see below and section 11.2.5).

```
panel.language=function(x, y, language)
{
   fit_language=function(lang_index, col_str)
   {
   points(x[lang_index], y[lang_index], col=pal_col[col_str])
   lines(loess.smooth(x[lang_index], y[lang_index], span=0.7), col=pal_col[col_str])
   }

fit_language(language == "Ada", 2)
fit_language(language != "Ada", 1)
}

# rows 28 and 30 are zero, and we only want columns 16:19
pairs(log(nasa[-c(28, 30) , 16:19]), cex.labels=2.0,
                panel=panel.language, language=nasa$language)
```

The default behavior of `pairs` produces a plot containing redundant information; it is possible to display different information in the upper and lower halves of the plot, and along the diagonal. Figure 8.7 shows expert and novice performance (time taken to complete various tasks and final test coverage) in a test driven development task, with a boxplot along the diagonal and correlation between each pair of attributes, for the two kinds of subjects, in the lower half of the plot. This plot, which primarily uses the default values for its visual appearance, needs more work before being presented to customers.

```
panel_user=function(x, y, user)
{
expert=(user == "e")
points(x[expert], y[expert], col=pal_col[1])
points(x[!expert], y[!expert], col=pal_col[2])
}

panel_correlation=function(x, y, user)
{
expert=(user == "e")
r_ex=cor(x[expert], y[expert])
r_nov=cor(x[!expert], y[!expert])
txt = paste0("e= ", round(r_ex, 2), "\n", "n= ", round(r_nov, 2))
text(0.0, 0.5, txt, pos=4, cex=1.6)
}

panel_boxplot=function(x, user)
{
```

```
t=data.frame(x, user)
boxplot(x ~ user, data=t, notch=TRUE, border=pal_col, add=TRUE)
}

pairs( ~ duration.min+changes+TDD+
          log(development.cycle.length)+line.coverage+block.coverage,
        data=tdd, cex.labels=1.3,
        upper.panel=panel_user, lower.panel=panel_correlation,
        diag.panel=panel_boxplot, user=tdd$user)
```

The splom function in the lattice package supports creating more complex pair-wise plots.

As the number of columns increases, the amount of detail visible in a pairs plot decreases. The correlation between pairs of columns can be compactly displayed, and provides the minimally useful information (i.e., a linear relationship exists).

The corrgram package implements various techniques for displaying correlation information. Figure 8.8 shows the correlation between every pair of 27 columns, with correlation used to control the color of each entry. In the upper triangle blue/clockwise denotes a positive correlation, and red/anti-clockwise a negative one; in the lower triangle the numeric values are also blue/red colored; looking at the numbers, more reader effort is needed to locate pairs having a high correlation (coloring reduces the effort).

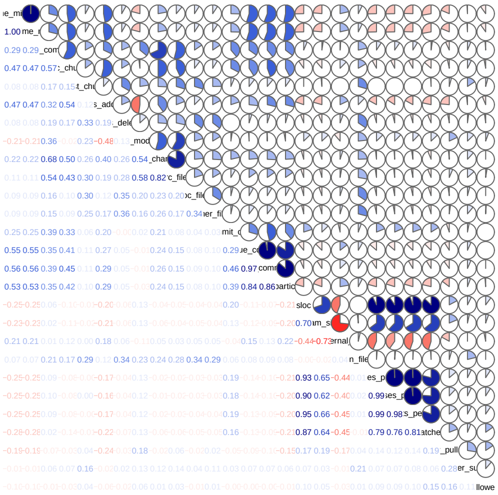


Figure 8.8: Correlations between pairs of attributes of 12,799 Github pull requests to the Homebrew repo, represented using numeric values and pie charts. Data from Gousios et al.[711] Github–Local

Having column names appear along the diagonal creates a compact plot; when many columns are involved this form of display is better suited to situations where the names follow a regular pattern. The plotcorr function in the ellipse package places column names around the outside; see Github–communicating/pull-req-cor.R.

```
library("corrgram")

corrgram(ctab, upper.panel=panel.pie, lower.panel=panel.shade)
```

Hierarchical clustering is another technique for finding columns that share some degree of similarity, based on a user supplied distance metric. The hclust function requires the user to handle the details; the varclus function in the Hmisc package provides a higher level interface. The following code uses as.dist to map the cross-correlation matrix returned by cor to a distance, to produce figure 8.9:

```
library("dendextend") # for coloring effects

# Cross correlation
ctab = cor(used, method = "spearman", use="complete.obs")

pull_dist=as.dist((1-ctab)^2)
t=as.dendrogram(hclust(pull_dist), hang=0.2)

col_pull=color_labels(t, k=5)
col_pull=color_branches(col_pull, k=2)
plot(col_pull, main="", sub="", col=point_col, xlab="", ylab="Height\n")
mtext("Pull related variables", side=1, padj=14, cex=0.7)
```

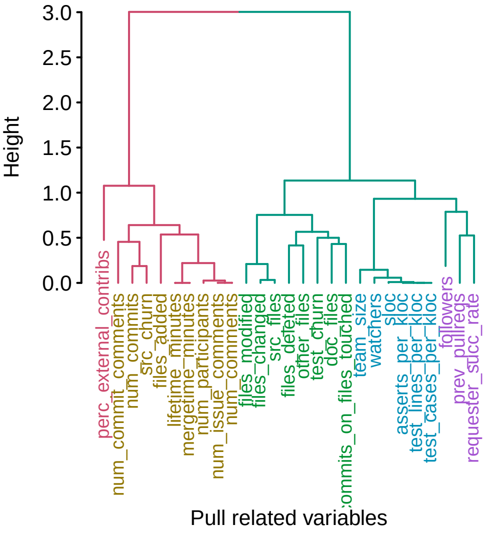


Figure 8.9: Hierarchical cluster of correlation between pairs of attributes of 12,799 Github pull requests to the Homebrew repo. Data from Gousios et al.[711] Github–Local

8.2.2 Guiding the eye through data

It may be difficult to reliably estimate the path of a central line through a collection of points, in a plot (according to some other goodness of fit criteria; section 11.2.5 contains a more detailed discussion of fitting a trend line to data). A way to quickly add such a line to an existing plot is to use the loess.smooth function, with the following code producing figure 8.10:

```
plot(res_tab$Var1, res_tab$Freq, col=pal_col[2],
        xlab="SPECint result", ylab="Number of computers\n")

lines(loess.smooth(res_tab$Var1, res_tab$Freq, span=0.3), col=pal_col[1])

scatter.smooth(res_tab$Var1, res_tab$Freq, span=0.3, col=point_col,
        xlab="SPECint Result", ylab="Number of computers\n")
```

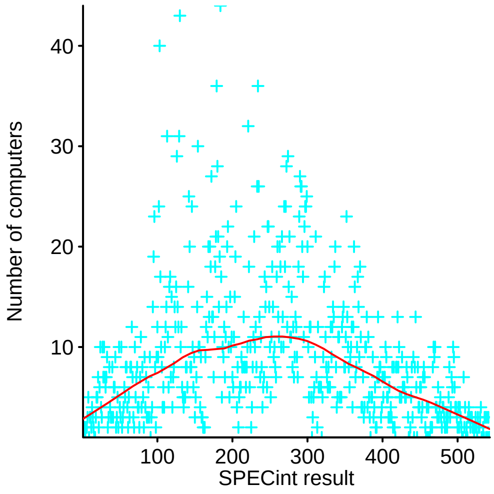


Figure 8.10: Number of computers having a given SPECint result; line is a loess fit. Data from SPEC.[1720] Github–Local

The `scatter.smooth` function both plots and draws the loess line (no options are available to control the color of the line).

Plotting a fitted line is a way of visually showing that expectations of a pattern of behavior is being followed (or not). Figure 8.11 shows a loess fit (green) to NASA data[742] on cost overruns for various space probes, against effort invested in upfront project definition; the upward arrow shows the continuing direction of the line seen in the original plot created by one user of this data (who was promoting a message that less investment is always bad).

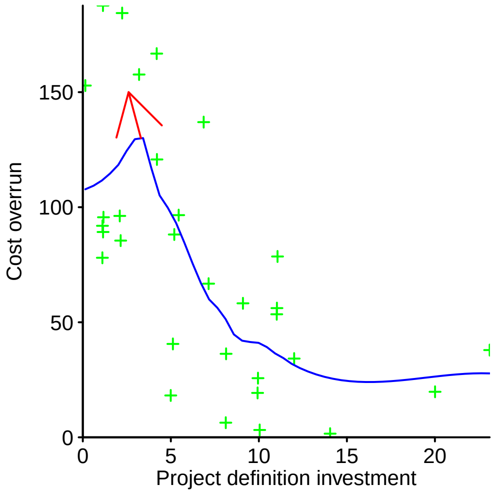


Figure 8.11: Effort invested in project definition (as percentage of original estimate) against cost overrun (as percentage of original estimate). Data extracted from Gruhl.[742] Github–Local

There are a variety of techniques for calculating a smooth line that is visually less noisy than drawing a line through all the points. Splines are invariably suggested in any discussion of fitting a smooth curve to an arbitrary set of points; the `smooth.spline` function will fit splines to a series of points and return the x/y coordinates of the fitted curve.

Splines originated as a method for connecting a sequence of points by a visually attractive smooth curve, not as a method of fitting a curve that minimises the error in some measurement. LOESS is a regression modeling technique for fitting a smooth curve that minimises the error between the points and the fitted curve; the `loess.smooth` function fits a loess model to the points and return the x/y coordinates of the fitted curve.

Both splines and loess can be badly behaved when asked to fit points that include extreme outliers, or have regions that are sparsely populated with data. The running median (e.g., `median(x[1:k])`, `median(x[(1+1):(k+1)])`, `median(x[(1+2):(k+2)])` and so on for some `k`) is a smoothing function that is robust to outliers; the `runmed` function calculates the running median of the points and returns these values (the points need to be in increasing, or decreasing, order).

Figure 8.12 shows the relative clock frequency of cpus introduced between 1971 and 2010; the various lines were produced using the values returned by the `smooth.spline`, `loess.smooth` and `runmed` functions (also see fig 14.4). Don't be lulled into a false sense of security by the lines looking very similar, the *smoothing parameter* provided by each function was manually selected to produce a visually pleasing fit in each case; the mathematics behind the functions can produce curves that look very different, and the choice of function will depend on the kind of curve required and perhaps be driven by the characteristics of the data.

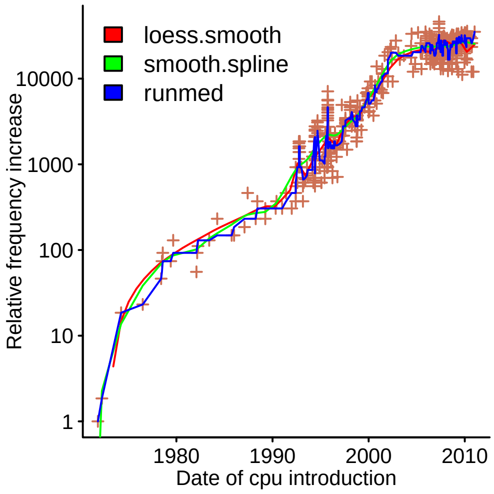


Figure 8.12: Relative clock frequency of cpus when first launched (1970 == 1). Data from Danowitz et al.[429] Github–Local

```
plot(x_vals, y_vals, log="y", col=point_col,
        xlab="Date of cpu introduction", ylab="Relative frequency increase\n")
lines(loess.smooth(x_vals, y_vals, span=0.05), col=pal_col[1])

# smooth.spline and runmed don't handle NAs
t=!is.na(x_vals) ; x_vals=x_vals[t] ; y_vals=y_vals[t]
t=!is.na(y_vals) ; x_vals=x_vals[t] ; y_vals=y_vals[t]

lines(smooth.spline(x_vals, y_vals, spar=0.7), col=pal_col[2])

t=order(x_vals)
lines(x_vals[t], runmed(y_vals[t], k=9), col=pal_col[3])
```

Lines drawn through a sample of measurements values often follow the path specified by a central location metric, e.g., the mean value. In more cases it may be more informative to fit a line such that 25% of measurements are below/above it, or some other percentage; *quantile regression* is a popular technique used for fitting such lines. Figure 8.13 is based on a study[1574] of 2,183 replies from a survey of FLOSS developers; two questions being the year and age at which those responding first contributed to FLOSS.

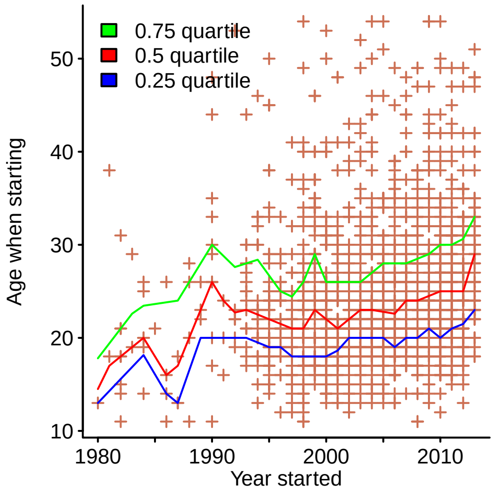


Figure 8.13: Year and age at which survey respondents started contributing to FLOSS, i.e., made their first FLOSS contribution. Data from Robles et al.[1574] Github–Local

If you find yourself writing lots of algorithmic R code during initial data exploration, you are either investing too much effort in one area, or you have found what you are looking for and have moved past initial exploration. Why are you writing lots of R, there is probably a package that does most of what you want to do and perhaps even more.

8.2.3 Smoothing data

Measured values sometimes fluctuate widely around a general trend (the data is said to be *noisey*). Smoothing the data can make it easier to see any pattern that might be present in the clutter of measured values. The traditional approach is to divide the range of measurement values into a sequence of fixed width bins and count the number of data points in each bin; the plotted form of this binning process is known as a *histogram*.

Histograms have the advantage of being easy to explain to people who do not have a mathematical background and existing widespread usage means that readers are likely to have encountered them before. Until the general availability of computers, histograms also had the advantage of keeping the human effort needed to smooth data within reasonable limits.

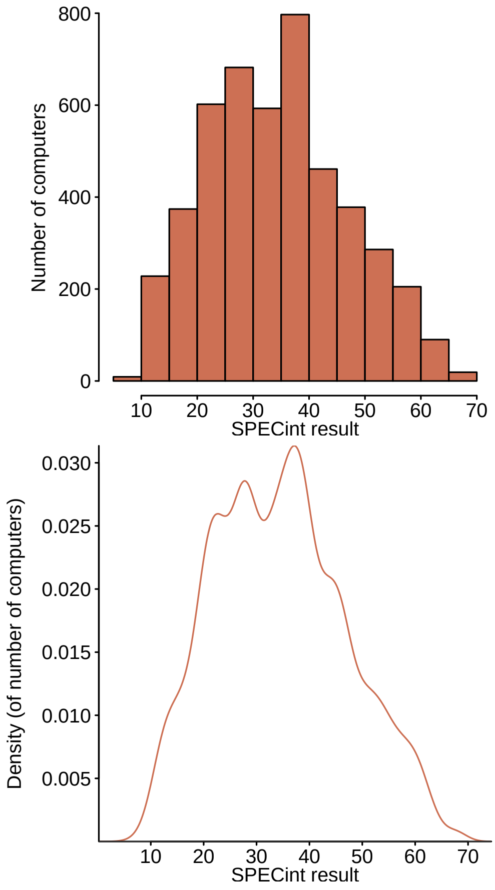


Figure 8.14: Number of computers with a given SPECint result, summed within 13 equal width bins (upper) and kernel density plot (lower). Data from SPEC.[1720] Github–Local

Figure 8.14, upper plot, shows a count of computers having the same SPECint result, aggregated into 13 fixed width bins (the number of bins selected by the hist function for this data).

```
hist(cint$Result, main="", col=point_col,
        xlab="SPECint result", ylab="Number of computers\n")
```

The histogram package supports a wider range of functionality and more options than is available in the base system functions.

The advantage of the binning approach to smoothing and aggregating data is ease of manual implementation, and for this reason it has a long history. The disadvantages of histograms are: 1) changing the starting value of the first bin can dramatically alter the visual outline of the created histogram, and 2) they do not have helpful mathematical properties.

A technique that removes the arbitrariness of histogram bins' starting position is averaging over all starting positions, for a given bin width (known as a *average shifted histogram*); this is exactly the effect achieved using kernel density with a rectangular kernel function.

It often makes sense for the contribution made by each value to be distributed across adjacent measurement points, with closer points getting a larger contribution than those further away. This kind of smoothing calculation is too compute-intensive to be suited to manual implementation, but are easily calculated when a computer is available.

The distribution of values across close measurement points is known as *kernel density*; histograms are the manual labourer's poor approximation to Kernel density, if a computer is available use the better technique.

The density function returns a kernel density estimate (which can be passed to plot or lines); the following code produced the lower plot in figure 8.14:

```
plot(density(cint$Result))
```

Density plots also perform well when comparing rapidly fluctuating measurements of related items. Figure 8.15, upper plot, shows the number of commits of different lengths (in lines of code) to the Linux filesystem code, for various categories of changes; the lower plot is a density plot of the same data.

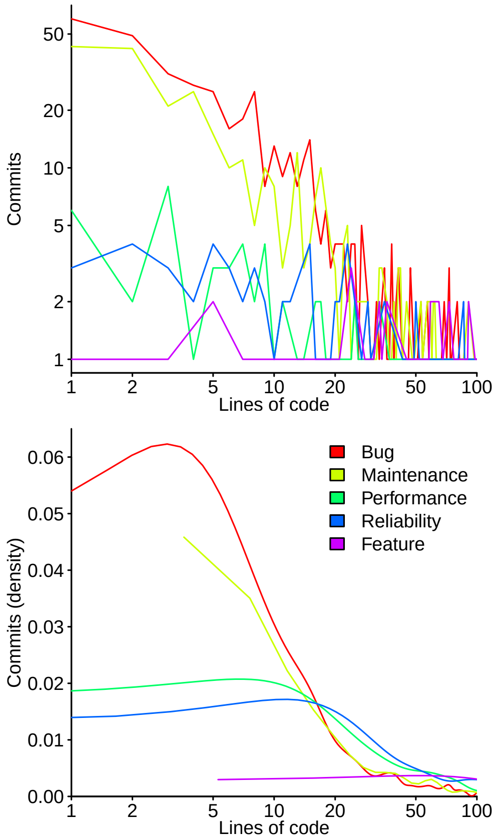


Figure 8.15: Number of commits containing a given number of lines of code made when making various categories of changes to the Linux filesystem code (upper), and a density plot of the same data (lower). Data from Lu et al.[1152] Github–Local

The kernel density approach generalizes to more than one dimension; see the KernSmooth and ks packages.

When dealing with measurements that span several orders of magnitude, a log scale is often used. Creating a histogram using a log scale requires the use of bin widths that grow geometrically (coding is needed to get the hist function to use variable width bins; the histogram package contains built-in support for this functionality), and bin contents has to be expressed as a density (rather than a count). A histogram based on counts, rather than density, can produce misleading results; figure 8.16 was produced by the following code, where y is assigned decreasing values (the histogram should be continuously decreasing and not show a second peak, which is an artifact generated by inappropriate analysis):

```
x=1:1e6
y=trunc(1e6/x^1.5)
log_y=log10(y)

hist(log_y, n=40, xlim=c(0, 3),
        main="", xlab="log(quantity)", ylab="Count\n")
```

8.2.4 Densely populated measurement points

Some samples contain data whose characteristics produce result in plots containing lots of ink and little visual information; some common characteristics of the density of values include:

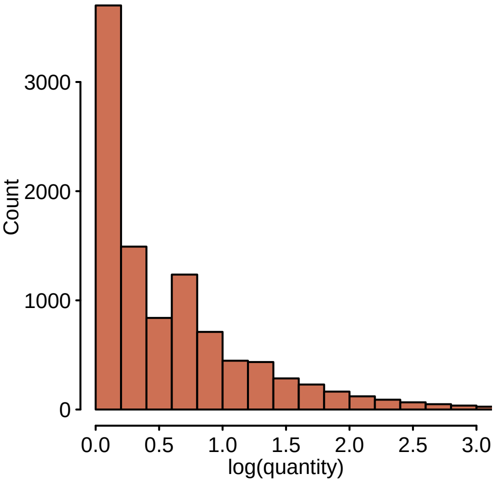


Figure 8.16: Histogram of the log of some measured quantity. Github–Local

- adjacent values on the x-axis having widely different values on the y-values, e.g., figure 8.10,
- multiple points having the same x/y value, all combined visually as a single point in a plot, e.g., figure 8.17,
- many very similar values that merge into a formless mass, when plotted, e.g., figure 8.18.

A plot of values gives a misleading impression when multiple measurements have the same value, i.e., a single point represents many measurements (the problem is more likely to occur for measurements that can only take a small set of values, e.g., discrete values); see figure 8.17, upper plot. The jitter function returns its argument with a small amount of added random noise; the middle plot of figure 8.17 shows the effect of jittering the values used in the upper plot. Another possibility is for the size of the plotted symbol to vary with the number of measurements at a given point (see figure 8.17, lower plot); as discussed elsewhere, people are poor at estimating the relative area and so size should not be treated as anything more than a rough indicator.

```
plot(maint$est_time, maint$act_time, col=point_col, xlab="",
                                         ylab="Actual hours\n")

plot(jitter(maint$est_time), jitter(maint$act_time), col=point_col,
                xlab="Estimated hours", ylab="Actual hours\n")

library("plyr")
t=ddply(maint, .(est_time, act_time), nrow)
plot(t$est_time, t$act_time, cex=log(1+t$V1), pch=1, col=point_col,
                                xlab="", ylab="Actual hours\n")
```

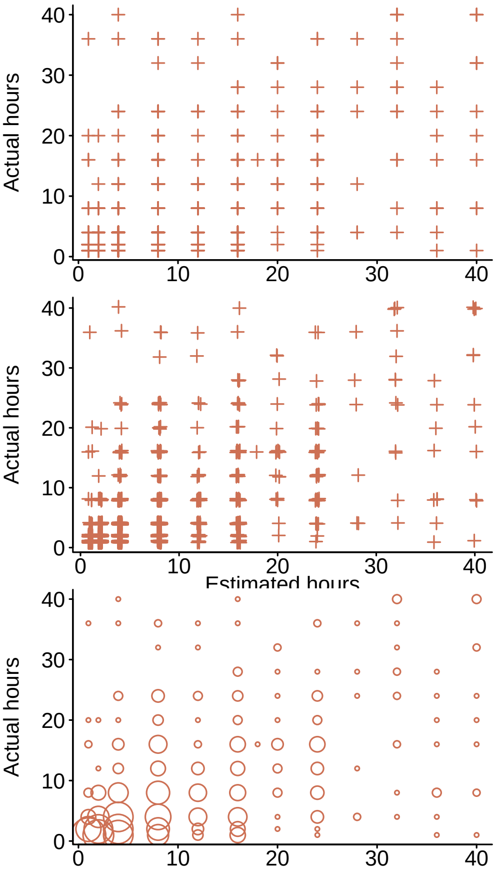


Figure 8.17: Developer estimated effort against actual effort (in hours), for various maintenance tasks, e.g., adaptive, corrective and perfective; upper as-is, middle jittered values and lower size proportional to the log of the number measurements. Data from Hatton.[779] Github–Local

A different kind of communications problem occurs when data points are so densely packed together, that any patterns that might be present are hidden by the visual uniformity (figure 8.18, upper plot; also see fig 11.23). One technique for uncovering patterns in what appears to be a uniform surface is to display the density of points. The smoothScatter function calculates a kernel density over the points to produce a color representation (middle plot); contour lines can be drawn with contour using the 2-D kernel density returned by kde2d (lower plot).

```
plot(udd$age, udd$insts, log="y", col=point_col,
                xlab="Age (days)", ylab="Installations\n")
# Bug in support for log argument :-(
smoothScatter(udd$age, log(udd$insts),
                xlab="Age (days)", ylab="log(Installations)\n")

library("MASS")

plot(udd$age, udd$insts, log="y", col=point_col,
        xlab="Age (days)", ylab="Installations\n")

# There is no log option, so we have to compress/expand ourselves.
d2_den=kde2d(udd$age, log(udd$insts+1e-5), n=50)
contour(d2_den$x, exp(d2_den$y), d2_den$z, nlevels=5, add=TRUE)
```

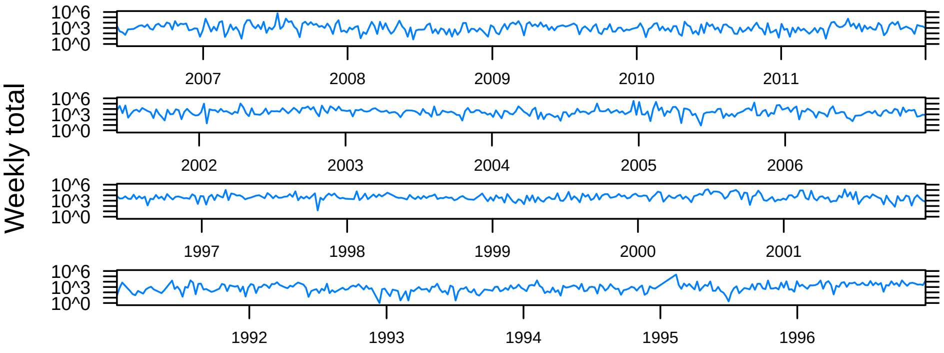


Figure 8.19: Number of lines added to glibc each week. Data from González-Barahona et al.[696] Github–Local

The hexbin package is available for those who insist on putting values into bins, in this case using hexagonal binning to support two dimensions.

One solution to a high density of points in a plot, is to stretch the plot over multiple lines; the xyplot function, in the lattice package, can produce a strip-plot such as the one in figure 8.19, produced by the following code:

```
library("lattice")
library("plyr")

cfl_week=ddply(cfl, .(week),
                function(df) data.frame(num_commits=length(unique(df$commit)),
                                        lines_added=sum(df$added),
                                        lines_deleted=sum(df$removed)))

# Placement of vertical strips is sensitive to the range of values
# on the y-axis, which may have to be compressed, e.g., sqrt(...).
t=xyplot(lines_added ~ week | equal.count(week, 4, overlap=0.1),
                cfl_week, type="l", aspect="xy", strip=FALSE,
                xlab="", ylab="Weekly total",
                scales=list(x=list(relation="sliced", axs="i"),
                            y=list(alternating=FALSE, log=TRUE)))
plot(t)
```

8.2.5 Visualizing a single column of values

The available data may contain a single column of values, or only one column of interest, i.e., there is no related column that can be used to create a 2-D plot. A *box-and-whiskers* plot (or *boxplot* as it is more generally known) is a traditional visualization technique that is practical to perform manually. Figure 8.20 highlights the following characteristics:

- median, i.e., the point that divides the number of values in half,
- first/third or lower/upper quartile, the 25th/75th percentiles respectively,
- lower/upper hinges, the points at a distance $\pm 1.5 \cdot IQR$ where IQR is the interquartile range (the difference between the lower quartile and the upper quartile). The dotted line joining the hinges to the quartile box are the whiskers,
- outliers, all points outside the range of the lower/upper hinge.

The boxplot function produces a boxplot; the argument notch=TRUE can be used to create a plot that includes a *notch* indicating the 95% confidence interval of the median (right boxplot in figure 8.20).

```
box_inf=boxplot(eclipse_rep$min.response.time, log="y",
                        boxwex=0.25, col="yellow", yaxt="n",
                        notch=TRUE, xlim=c(0.9, 1.3), ylab="")
```

When a computer is available to do the calculation, more visually informative techniques can be used. What is known as a *violin plot* uses a kernel density of the values, as the outline of the container image; see figure 8.21 (a mirror image is usually included in the plot, hence the name). The vioplot function in the vioplot package is used for the violin plots in this book.

```
library("vioplot")

vioplot(log(eclipse_rep$min.response.time), col="yellow", colMed="red",
        ylim=range(log(eclipse_rep$min.response.time)),
        xlab="", ylab="log(Seconds)")
```

Formula notation can be used to display multiple violin plots in the same plot, with the following code producing figure 8.22:

```
vioplot(time ~ group+task, data=gs, horizontal=TRUE, col=pal_col,
                xlab="Time (minutes)", ylab="")
```

A bar chart with error bars is regularly used to visually summarise values (sometimes known as *dynamite plots*). A study[396] investigating the effectiveness of various ways of visually summarizing data (including boxplots, violin plots and others) found that when extracting information from bar charts (with or without error bars) subjects did not perform as well as they did when using the other techniques.

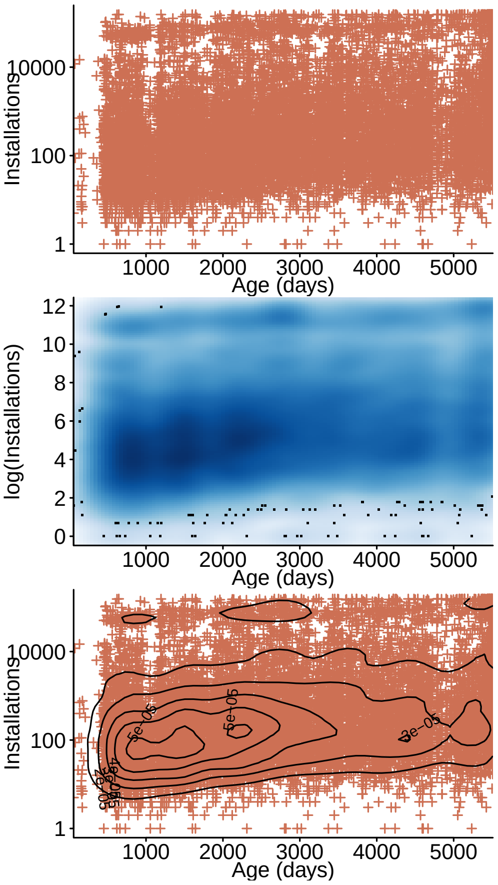


Figure 8.18: Number of installations of Debian packages against the age of the package; middle plot was created by smoothScatter and lower plot by contour. Data from the "wheezy" version of the Ultimate Debian Database project.[1842] Github–Local

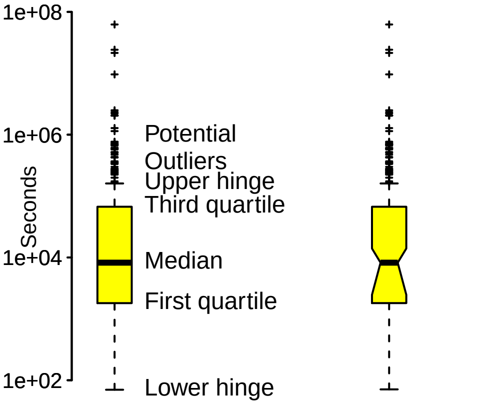


Figure 8.20: Boxplot of time between a potential mistake in Eclipse being reported and the first response to the report; right plot is notched. Data from Breu et al.[247] Github–Local

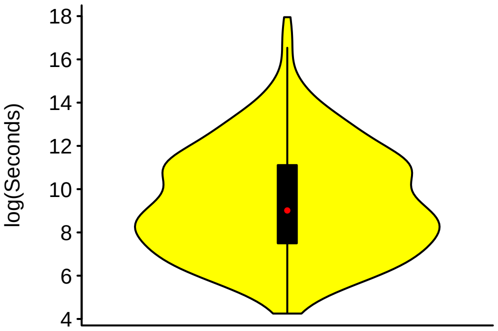


Figure 8.21: Violin plot of time between bug being reported in Eclipse and first response to the report. Data from Breu et al.[247] Github–Local

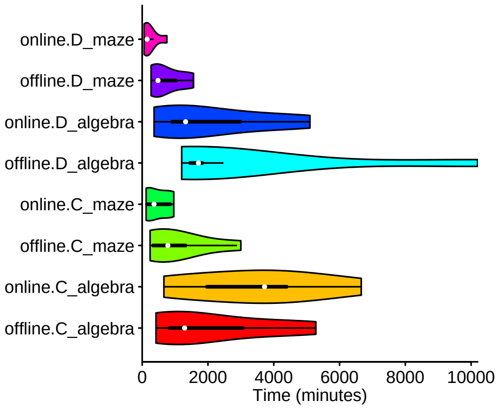


Figure 8.22: Time taken for developers to debug various programs using batch processing or online (i.e., time-sharing) systems. Data kindly provided by Prechelt.[1502] Github–Local

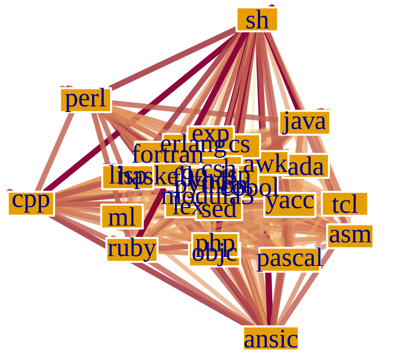


Figure 8.23: Pairs of languages used together in the same GitHub project with connecting line width, color and transparency related to number of occurrences. Data kindly supplied by Bissyande.[200] Github–Local

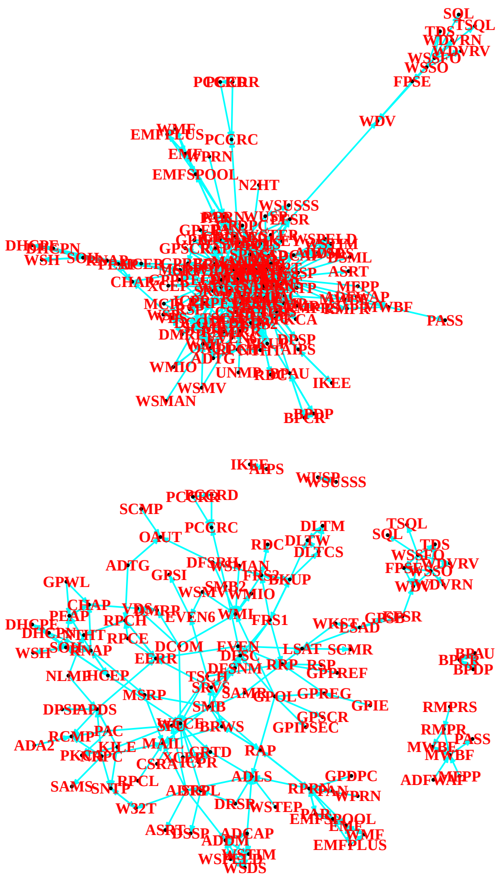


Figure 8.24: References from one document to another in the Microsoft Server Protocol specifications. Data extracted by your author from the 2009 document release.[1260] Github–Local

8.2.6 Relationships between items

The relationship between two entities may be the attribute of interest. Graphs are the data structure commonly associated with relationships, and the `igraph` package contains numerous functions for processing graphs.

When displaying graphs containing large numbers of nodes, potentially useful information in the visual presentation may be swamped by many nodes having relatively few connections. Figure 8.23 is an attempt to show which languages commonly occur, within the same project, with another language, in a sample of 100,000 GitHub projects. The number of projects making use of a given pair of languages is represented using line width and to stop the plot being an amorphous blob the color and transparency of lines also changes with number of occurrences.

Perhaps items having relatively few connections are the ones of interest. The Microsoft Server protocol specifications[1260] contain over 16 thousand pages, across 130 documents (the client specification documents are also numerous). Figure 8.24, upper plot, shows dependencies between the documents (based on cross-document references in the 2009 release[1260]); the lower plot shows the dependencies after excluding the 18 most referenced documents (plot based on the following code):

```
library("igraph")
library("sna")

interest_gr=graph.adjacency(interest, mode="directed")

# V(interest_gr)[names(in_deg)]$size=3+in_deg^0.7
V(interest_gr)$size=1
V(interest_gr)$label.color="red"; V(interest_gr)$label.cex=0.75
E(interest_gr)$arrow.size=0.2

plot(interest_gr)
```

It is possible to use R to draw presentable graphs, however, if your primary interest is drawing visually attractive graphs containing lots of information, then there other systems that may be easier to use (e.g., GraphViz[720]). Yes, an R interface to these systems may be available, but if statistical analysis is not the primary purpose, why is R being used?

Alluvial plots are a method for visualizing the flow between connected entities. Figure 8.25 shows factors used to prioritize the application of Github pull requests, and the relative orders in which they appear in a dataset of pull requests;[712] the `alluvial` package was used.

8.2.7 3-dimensions

We live in a world of three spatial dimensions, which is only one more than the two dimensions available on flat screens and paper; various techniques for enhancing a flat surface to display information in one more dimension are available.

Heatmaps use color to display information about a third quantity within a 2-D plot. Figure 8.26 shows the L3 cache bandwidth (color+number) of an Intel Sandy Bridge processor running at various clock frequencies and using various combinations of cores.

Both the `heatmap` function in the base system and the `heatmap.2` function in the `gplots` package, clusters the rows/columns and then plots a dendrogram; various arguments have to be set to switch off this default behavior, with `heatmap` doing its best to make life difficult including not coexisting with other plots in the same image; `heatmap.2` is more reasonable.

The `levelplot` function in the `lattice` package provides straightforward functionality for producing heat maps, and it is used to produce all the heatmaps in this book.

```
library("lattice")

t=levelplot(L3_band,
                col.regions=rainbow(100, end=0.9),
                xlab="Clock frequency (Mhz)", ylab="Cores used",
                scales=list(x=list(cex=0.70, rot=35),
                            y=list(cex=0.65)),
                panel=function(...)
                        {
                        panel.levelplot(...)
                        panel.text(1:11, rep(1:8, each=11),
                                                L3_band, cex=0.55)
                        })

plot(t, panel.height=list(3.8, "cm"), panel.width=list(6.2, "cm"))
```

A contour plot can be used for visualizing the relationship between a response variable and two explanatory variables; the contour function is part of the base system. A study by Thereska, Doebel, Zheng and Nobel[1808] measured the performance of various applications running on a variety of desktop computers; the cpu speed and memory capacity of the computer hosting each of the 4,924,467 user sessions was recorded. The contours in figure 8.27 are based on the number of user sessions measured on computers having a given processor speed and memory capacity.

```
library("plyr")

Um=unique(memcpu$MemorySize)
M_map=mapvalues(memcpu$MemorySize, from=Um, to=rank(Um))

Us=unique(memcpu$ProcSpeed)
S_map=mapvalues(memcpu$ProcSpeed, from=Us, to=rank(Us))

cnt_mat=matrix(data=0, nrow=length(Us), ncol=length(Um))

cnt_mat[cbind(S_map, M_map)]=log(memcpu$Session_Count)

contour(x=seq(min(Us)/max(Us), 1, length.out=length(Us)),
        y=seq(min(Um)/max(Um), 1, length.out=length(Um)),
        z=cnt_mat, col=pal_col, nlevels=10, axes=FALSE,
        xlim=c(min(Us)/max(Us), 1), ylim=c(min(Um)/max(Um), 1),
        xlab="Processor speed (GHz)",
        ylab="Memory size (Mbyte)\n")

axis(1, at=sort(Us)/max(Us), labels=sort(Us))
axis(2, at=sort(Um)/max(Um), labels=sort(Um))
```

A variety of functions are available for representing a 3-D plot as on 2-D surface, including the scatterplot3d function in the car, and the plot3d function in the rgl package.

Histograms in 3-dimensions provide more opportunities, than histograms in 2-dimensions, for looking impressive with little data and misleading viewers. A study by Hamill and Goseva-Popstojanov[766] investigated the origin of 1,257 faults in 21 large safety critical applications, recording where the fixes were made (e.g., requirements, design, code or supporting files). Figure 8.28 shows a 3-D histogram of root cause/fix location on the x-y axis and a count of occurrences on the z-axis. Color has the effect of enhancing the visual appearance of the plot, and makes it easier to locate stacks having similar values, but it is very difficult to obtain detailed information from this plot. Adding numeric values would provide detail, but the real issue is what information is the plot intended to communicate?

```
library("lattice")
library("latticeExtra")

# log transform pulls out small differences in majority of counts
transform_breaks= exp(do.breaks(range(log(1e-4+STVR_col$occurrences)), 20))
t=cloud(occurrences ~ fix+fault, STVR_col,
                panel.3d.cloud=panel.3dbars,
                xlab="Fixes involved", ylab="Fault found", zlab="Count",
                xbase=0.5, ybase=0.5, aspect=c(1, 1),
                col.facet = level.colors(STVR_col$occurrences,
                                at = transform_breaks,
                                col.regions = rainbow),
                scales=list(arrows=FALSE, distance=c(2, 1.1, 1),
                        x=list(rot=-20) # Rotate tick labels
                        ))
plot(t)
```

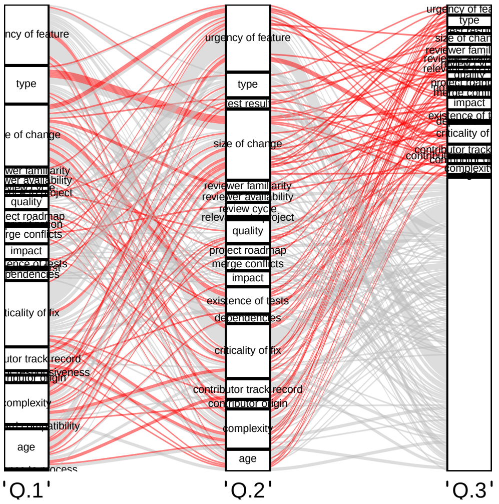
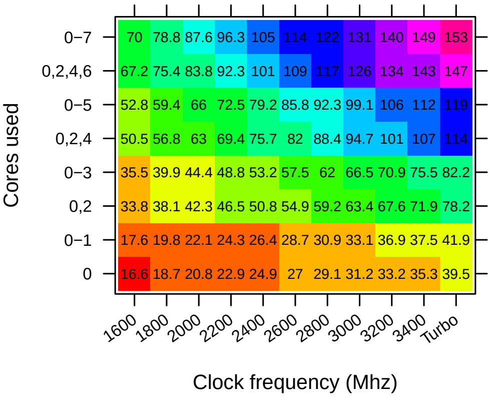


Figure 8.25: Alluvial plot of relative prioritization order of selection and application of Github pull requests. Data from Gousios et al.[712] Github–Local



	1600	1800	2000	2200	2400	2600	2800	3000	3200	3400	Turbo
0−7	70	78.8	87.6	96.3	105	114	122	131	140	149	153
0,2,4,6	67.2	75.4	83.8	92.3	101	109	117	126	134	143	147
0−5	52.8	59.4	66	72.5	79.2	85.8	92.3	99.1	106	112	119
0,2,4	50.5	56.8	63	69.4	75.7	82	88.4	94.7	101	107	114
0−3	35.5	39.9	44.4	48.8	53.2	57.5	62	66.5	70.9	75.5	82.2
0,2	33.8	38.1	42.3	46.5	50.8	54.9	59.2	63.4	67.6	71.9	78.2
0−1	17.6	19.8	22.1	24.3	26.4	28.7	30.9	33.1	36.9	37.5	41.9
0	16.6	18.7	20.8	22.9	24.9	27	29.1	31.2	33.2	35.3	39.5

Figure 8.26: Intel Sandy Bridge L3 cache bandwidth in GB/s at various clock frequencies and using combinations of cores (0-3 denotes cores zero-through-three, 0,2,4 denotes the three cores: zero, two and four). Data from Schone et al.[1628] Github–Local

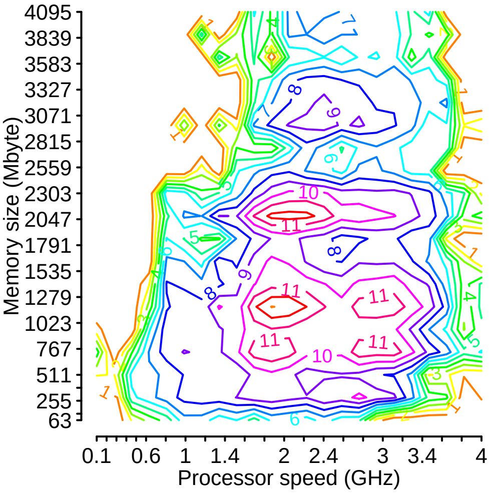


Figure 8.27: Contour plot of number of sessions executed on a computer having a given processor speed and memory capacity. Data kindly provided by Thereska.[1808] Github–Local

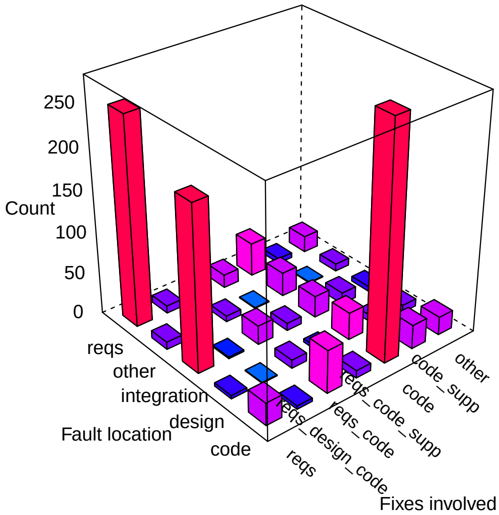


Figure 8.28: Root source of 1,257 faults and where fixes were applied for 21 large safety critical applications. Data from Hamill et al.[766] Github–Local

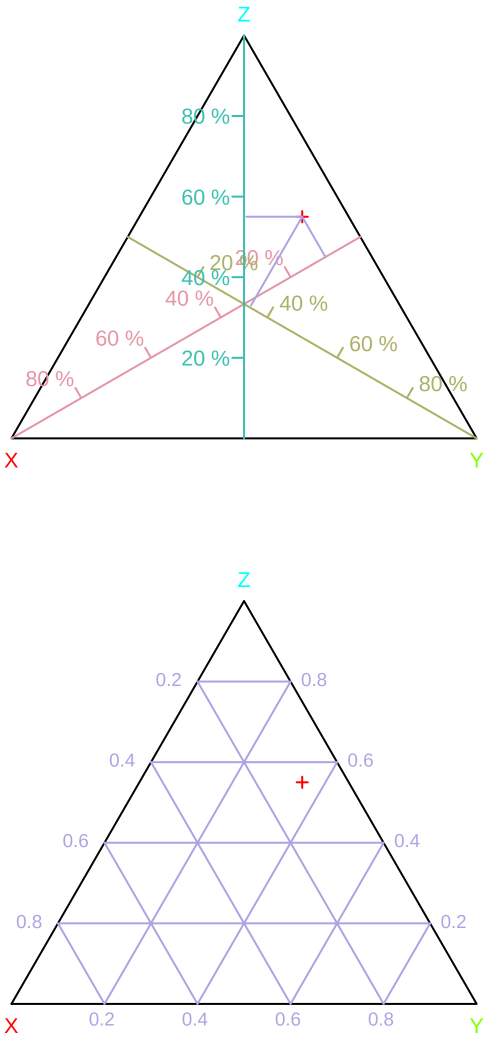


Figure 8.29: Ternary plots drawn with two possible visual aids for estimating the position of a point (red plus at x=0.1, y=0.35, z=0.55); axis names appear on the vertex opposite the axis they denote. Github–Local

A ternary, or triangle, plot has three axes. The axes are inclined at an angle of 60°to each other, and practice is needed to become proficient at estimating the coordinates of any point. Figure 8.29 shows two ways of labelling a ternary plot (with the three coordinates summing to 100%), with labels appearing at the vertex rather than along the axis and axis scales drawn either perpendicular to the axis or labeled along the axis and within the triangle as a grid. The upper plot shows how lines perpendicular to the appropriate axis are used to find the location of a point (at 10, 35, 55 in this case).

The closer points are to a vertex the larger the value of the corresponding variable, the closer points are to an axis the smaller the value of the corresponding variable.

Ternary plots are used to visualize compositional data (see fig 5.32); the compositions and vcd packages include support for creating ternary plots.

In the following code rcomp normalises its argument (so that rows sum to 100) using an interval scale and returns an object having class rcomp (the compositions package has overloaded functions for handing objects of this type):

```
library("compositions")

xyz=c(10, 35, 55)
plot(rcomp(xyz), labels="", col="red", mp=NULL)
ternaryAxis(side=-1:-3, labels=paste(seq(20, 80, by=20), "%"),
                pos=c(0.5,0.5,0.5), col.axis=hcl_col, col.lab=pal_col,
                small=TRUE, aspanel=TRUE,
                Xlab="X", Ylab="Y", Zlab="Z")

lines(rcomp(rbind(xyz, c(10, 45, 45))), col=hcl_col[4])
lines(rcomp(rbind(xyz, c(32, 35, 33))), col=hcl_col[4])
lines(rcomp(rbind(xyz, c(22, 23, 55))), col=hcl_col[4])

plot(rcomp(xyz), labels="", col="red", mp=NULL)

isoPortionLines(col=hcl_col[4])
ternaryAxis(side=0, col.axis=hcl_col, small=TRUE, aspanel=TRUE,
                Xlab="X", Ylab="Y", Zlab="Z")
```

8.3 Communicating a story

Results from data analysis are of no value unless they are reliably communicated to the target audience.[354] Reliably communicating information to other people is difficult; the intended message may be misunderstood or important parts may simply be overlooked by readers. The data analyst has to provide a narrative that tells the intended story. How people process visual information is discussed in section 2.3.

No known algorithm is available that selects a method that ensures communication is made in a way that will be correctly interpreted by the audience.[i] The main techniques available for presenting numeric information, and how they might be implemented using R, are covered in the rest of this chapter.

There are a wide variety of ways of presenting information, e.g., tables, pie charts, bar charts and scatter plots. Which of these is best, at communicating information to readers? The answer from a wide range of studies is that it depends on what information readers are trying to obtain, and the following is a brief summary of some research findings:

- graph or table? Studies have found that except for reading-off specific values (and recall of these values later), subjects perform better with line graphs, than tables. However, while graphs have better performance when presenting a given perspective (e.g., by selection of the axis), tables may be preferable[236] when wanting to present data in a way that does not favour any one perspective on the data; it boils down to selecting the best cognitive fit,[1874]
- the ability of pie charts to communicate information has been questioned over the years.[411] A study[1723] comparing subject performance using pie charts, a horizontal

[i] Studies have found that people are much better at extracting certain kinds of information when it is presented in the form of frequency of occurrence rather than as a percentage.[670]

divided bar chart, a vertical bar charts and a table, found that except when direct magnitude estimation was required pie charts were comparable to bar charts, but for combinations of proportions pie charts were superior; a study[1697] comparing the three visual clues present in a pie chart (i.e., angle, area and circumference length) found that angle and area were poor methods of communicating information, and that circumference length was the best of three,

- adding a third dimension to a graph has been found to slow down reader performance,[815] i.e., subjects take longer to extract information and may be less accurate. The conclusion would appear to be, don't use three dimensions when two will do. While subjects have expressed a preference for using 3-D graphs to impress others, no studies have investigated whether they have this effect,
- a study by Jansen and Hornbæk[904] investigated the perceived relative size of bars and spheres. Subjects saw an image containing either two bars of different length, or two spheres of different size, and were asked to estimate the size ratio of the two objects. Figure 8.30 shows the actual and estimated bar and sphere ratios for each of the ten subjects, with fitted regression lines;[ii] grey line shows where estimate equals actual. The lower plot shows subjects consistently underestimating the ratio of sphere sizes.
- studies by Cleveland are often cited in R related publications: one study[365] asked subjects to make judgements about graphical information encoded in various ways[iii]; the results showed that accuracy of subjects' answers varied slightly between encoding methods, the ordering from most accurate to least accurate was: position along a common scale, positions along nonaligned scales, length, direction, angle, area, volume, curvature and shading, color saturation. Later studies[1723] suggest that the factors are not so well-defined, with some effects found to be influenced by the structure of the experiments, or performance with a particular encoding depending on the task subjects' performed.

  The thinking behind some layout details used by R's `plot` function are based on experimental work by Cleveland.[364] Although not explicitly stated the aim appears to have been to present data in a workman-like way that avoids the possibility of plotted data values being obscured by plot markings (e.g., tick marks).

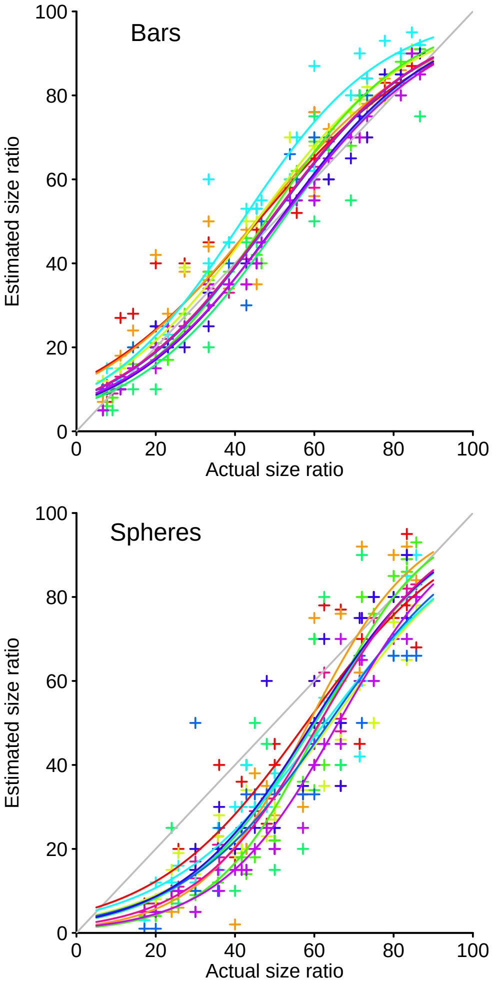


Figure 8.30: Actual and estimated size ratio for bars and spheres, for each of the ten subjects (in different colors, with line from fitted regression model), with grey line showing where estimate equals actual. Data from Jansen et al.[904] Github–Local

The `plot` function is a workhorse for handing the graphical display of data in R; it does a good job of producing a reasonable looking plot from whatever it is passed. Based on this book's implementation goal of using one implementation technique, where-ever possible, the plots in this book were generated using the `plot` function.

The `lattice`[1613] and `ggplot`[317] packages provide alternative world views on the plotting of data; `lattice` is based on the Trellis graphics system[156] from Bell Labs and has an emphasis on multivariate data, while the design of `ggplot` is derived from the work of Wilkinson.[1943iv] Both `lattice` and `ggplot` provide a great deal of control over the created plot through the use of user supplied functions. While `ggplot` is widely used by experienced R developers, its inability to sensibly handle whatever nonsense data is thrown at it (e.g., nonsense in that there are mistakes in the R code that produced it) prevents this package being recommended for casual use. A detailed technical overview R's graphics subsystems is available in "R Graphics" by Murrell.[1318]

Combining data with visual information familiar to readers helps them to extract patterns that mean something to them. Figure 8.31 shows single event upsets (i.e., radiation induced memory faults) experienced by NASA's Orbview-2 spacecraft during one day in 2000. Overlaying the satellite location at the time of the upset on a map of the Earth (using the `map` package) provides context to help readers understand where most upsets occur.

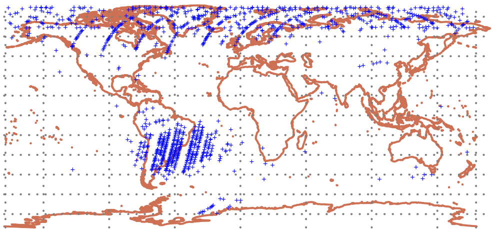


Figure 8.31: Earth relative positions of NASA's Orbview-2 spacecraft when it experienced a single event upset (in blue) on 12 July 2000. Data kindly provided by LaBel.[1480] Github–Local

In some cases the intent of a plot may be to communicate that life is complicated, or that there are a few big fish and many small ones. Figure 8.32 shows an estimate of the market share of Android devices in use in 2015, by brand/company and product name (based on the 682,000 unique devices that downloaded an App from OpenSignal[1402]). Treemaps encode information using area, a quantity that many readers have problems accurately interpreting.

```
library("treemap")
```

[ii] Beta regression is used, because it provides a much better fit to the data than the model (based on Stevens' power law) fitted[1722] by Jansen and Hornbæk.

[iii] One study[792] has replicated some of these findings.

[iv] The title of Wilkinson's book "The Grammar of Graphics" refers to the structure of software written to display graphics rather than the structure of the displayed information.

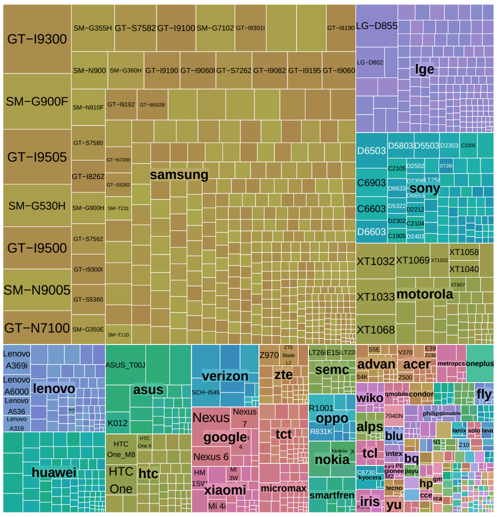


Figure 8.32: Estimated market share of Android devices by brand and product, based on downloads from 682,000 unique devices in 2015. Data from OpenSignal.[1402] Github–Local

```
and_tree=treemap(android, c("brand", "model"), "august2015",
              title="", palette=pal_col,
              border.col="white", border.lwds=c(0.5, 0.25))
```

8.3.1 What kind of story?

The kinds of output from statistical data analysis include the following:

- a description of the data, e.g., its mean and variance, how measurements cluster, an equation summarizing the data. A descriptive model, built from the data, can be used to help gain insights into the system that was measured, for comparing different descriptions (e.g., benchmark results) and for building similar systems (e.g., automatically creating file system contents[16] for benchmarking purposes)
- a model built to mimic the behavior of a system (as expressed in the measurements made), e.g., a simulator,
- a predictive model capable of making appropriately accurate predictions for values not in the set of measurements used to build the model. The possible range of prediction values may be within the range of values used to build the model or outside the range of these values, e.g., making predictions about a future time,

A standard reply to any complaints about the adequacy of a model built using data is the adage "All models are wrong, but some are useful."

An example of the different kinds of model that can be built, and how their usefulness depends on the problem they are intended to solve, is provided by a question involving the use of local variables in the source code of a function definition.

If the source code contains a total of N read accesses to variables defined locally within the function, what percentage of variables will be read from once, twice and so on (based on a static count of the visible source code, not a dynamic count obtained by executing the function)?

Data from an analysis of C source[919] provides a description of "what is". Plotting the data shows that a few variables account for most accesses (i.e., read from). After some experimentation the following equation was found to be a good fit to the data (see figure 8.33):

$pv = 34.2 \times e^{-0.26acc-0.0027N}$, where: pv is the percentage of variables, acc the number of read accesses to a given variable, and N is the total number of accesses to all local variables within a function. For example, when a function contains a total of 30 read accesses of its local variables, the expected percentage of variables accessed twice is: $34.2 \times e^{-0.26\times 2-0.0027\times 20}$.

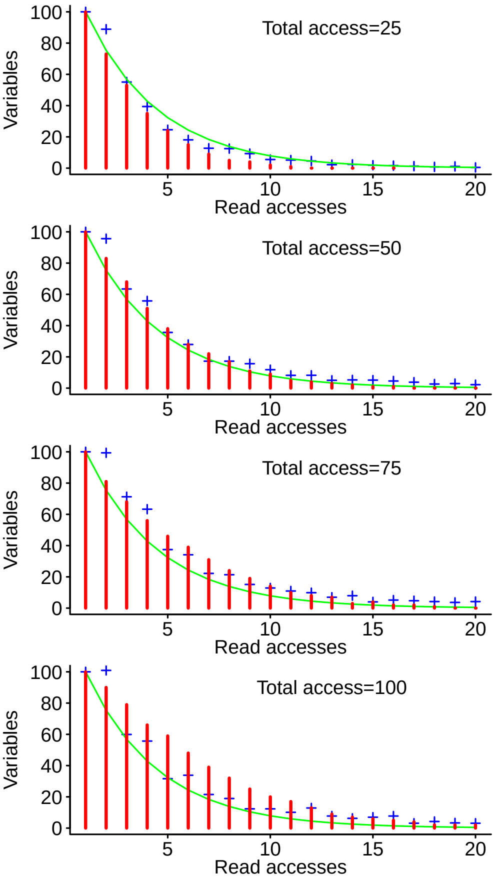


Figure 8.33: Variables having a given number of read accesses, given 25, 50, 75 and 100 total accesses, calculated from running the weighted preferential attachment algorithm (red), the smoothed data (blue), and a fitted exponential (green). Github–Local

What other kind of model can be built to answer this question?

This problem has a form that has parallels with the growth of new pages and links to existing pages on the world wide web. Each access of a local variable could be thought of as a link to the information contained in that variable. One idea that has been found to be integral to modeling the number of links between web pages is *Preferential attachment*.

With some experimentation an iterative algorithm based on preferential attachment, was created, that produced a pattern of behavior close to that seen in the data. The algorithm is as follows:

Assume we are automatically generating code for a function, and from the start of the function, to the current point in the code L distinct local variables exist (and have been accessed), with each accessed R_i times ($i = 1, \dots, L$). The following weighted preferential attachment algorithm is used to select the next local variable to access (global variables are ignored in this analysis):

- With probability $\frac{1}{1+0.5L}$ create a new variable to access,[v]
- with probability $1 - \frac{1}{1+0.5L}$ select a variable that has previously been accessed in the function, choose an existing variable with probability proportional to $R + 0.5L$ (where R is the number of times the variable has previously been read from;); e.g., if the total accesses up to this point in the code is 12, a variable that has had four previous read accesses is $\frac{4+0.5\cdot 12}{2+0.5\cdot 12} = \frac{10}{8}$ times as likely to be chosen as one that has had two previous accesses.

[v] The unweighted preferential attachment algorithm uses a fixed probability to decide whether to access a new variable.

The red points in figure 8.33 were calculated using the above algorithm.

This preferential attachment model provides insights into local variable usage that are very different from those provided by the fitted exponential equation. Neither of them could not be said to be realistic descriptions of the process used by developers, when writing code. Both models are descriptions of the end result of the emergent process of writing a function definition; each model has its own advantages and disadvantages, including the following:

- the fitted equation is fast and simple to calculate, while the output from the iterative model is slow (an average over 1,000 runs in the example code) and requires more work to implement,
- the iterative model automatically generates a possible sequence of accesses (for machine generated source), while a fitted equation does not provide any obvious method of generating a sequence of accesses,
- multiple executions of an iterative model can be used to obtain an estimate of standard deviation, while the equation does not provide a method for estimating this quantity (it may be possible to fit another regression model that provides this information),
- the equation provides an end-result way of thinking, while the iterative model provides a choice-based way of thinking about variable usage.

A common technique for devising a model for a new problem is to find a very similar problem that has a proven model, and to adapt this existing model to the new problem. A model based on existing practice is often easier to sell to an audience, than a completely new model.

Some multiprocessor system have a "shared nothing" architecture, which minimises the sharing of hardware resources. Performance measurements of such systems, under various loads, shows that even when tasks can be evenly distributed across all X processors in the system, performance is rarely X times faster. Which model provides a good explanation of the performance seen?

Amdahl's law predicts changes in multiprocessor performance as the number of processors used changes, where the multiprocessor system has a shared hardware architecture. Gunther[749] extended this "law" to cover multiprocessors having "shared nothing" architecture; the adapted model, plus a further adaption, are not good fits to the data.

Gunther[750] created a model based on queuing theory and simulated model performance (with each job waiting in a queue for time t_1 and executing for time t_2). The argument for using queuing theory is that data sharing between different programs can create resource contention that the "shared nothing" hardware architecture cannot unblock. Figure 8.34 shows that the queuing model more accurately follows the pattern measured. Given the small amount of data available it would be unwise to attempt further model tuning.

The R language does not contain features designed with simulation in mind[vi], but like most languages it can be adapted to solve problems outside its core domain; see the simFrame and simmer packages.

Finding a workable model, based on the available data, can involve many iterations over a long time. For instance, modeling the growth of the size and number of files/directories in a filesystem has a long history, with current models[1283] either involving a mixture of two distributions for the equation fitting approach, or a generative approach based on simulating the way new files are created from existing files.

Perhaps the most important question to answer when proposing any model is the purpose to which it will be put. A model intended to gain insight might not be of any use in making practical recommendations, and a model used to make predictions might not provide any useful insight. For instance, modeling the connection between modifications to files and the introduction of mistakes, which cause faults to be experienced may be used to predict coding mistake rates based on modification history, but such a model has limited scope for directly deriving methods for reducing faults (e.g., reduce faults by reducing file modifications, is of no use when customers want new or modified behavior in the applications they use).

In most cases, a great deal of domain knowledge is required to build a model having the desired level of performance. There is no guarantee that any created model will be sufficiently accurate to be useful for the problem at hand; this is a risk that occurs in all

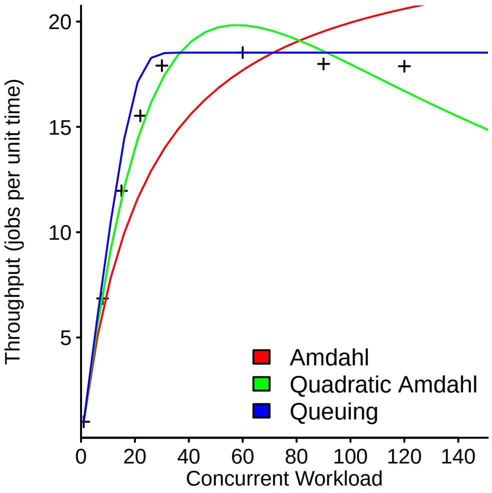


Figure 8.34: Throughput when running the SPEC SDM91 benchmark on a Sun SPARCcenter 2000 containing 8 CPUs, with the predictions from three fitted queuing models. Data from Gunther.[750] Github–Local

[vi] Interfacing to Netlogo[1507]

model building exercises. Ideally model building is driven by a theory describing the behavior of the system being modeled. When a theory is complete there is no need for new models, the fact that the creation of a new model is being considered implies that existing models are lacking in some respect.

8.3.2 Technicalities should go unnoticed

The machinery of information presentation should not get in the way of reader's access to that information.

There are many books offering tips, suggests and recommendations for how best to present visual information; the only book recommended by your author (it is based on a wide range of empirical research) is "Graph Design for the Eye and Mind" by Stephen Kosslyn.[1027] Sometimes multiple plots are used to tell an evolving story, McCloud[1215] is a great introduction to this art form.

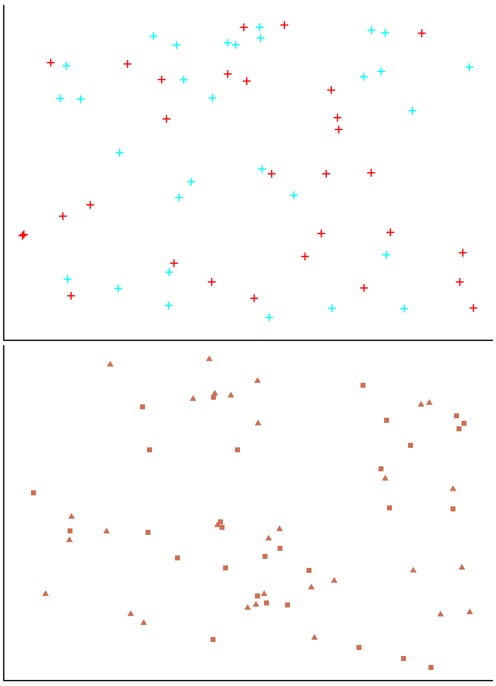


Figure 8.35: Illustration of the difference in cognitive effort needed to locate points differing by shape or color (one is a serial search, while the other operates in parallel). Github–Local

8.3.2.1 People have color vision

Until the mid 1980s most people used computer terminals that were only capable of displaying black and white (or green and black). Forty years later, the look-and-feel of mid-1980s computer usage still predominates in serious works of data visualization.

This book treats color as an essential component of numeric story telling. Color provides an extra dimension that enables more information to be present within the same area, and make it easier for viewers to extract information from a plot.[vii]

Selecting the most appropriate colors to use requires skill and experience. The `colorspace` and `RColorBrewer` packages both include functions that automatically select a color palette based on the arguments passed;[viii] the `colorspace` package provides a wider range of functionality than `RColorBrewer`, and is used to select the colors for the plots appearing in this book.

The Hue-Chroma-Luminence (HCL) color space is claimed[1982] to provide a better mapping to the human color perceptual system (hue: dominant wavelength; chroma: colorfulness, intensity compared to gray; and luminence: brightness, amount of gray), than alternative spaces.[ix] The color palettes generated by the `rainbow_hcl` function are considered to be qualitative palettes, that are suitable for depicting different categories; those generated by the `sequential_hcl` function to be suitable for coding numerical information that ranges over a given interval, with the `diverge_hcl` function also encoding numerical information, but including a neutral value.

The `choose_palette` function provides an interactive, slider based, method for developers to define their own color palettes.

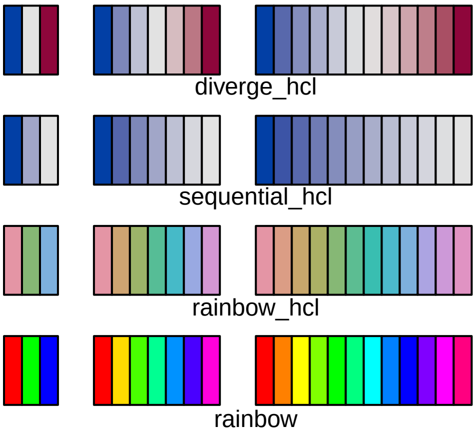


Figure 8.36: The three, seven and twelve color palettes returned by calls to the `diverge_hcl`, `sequential_hcl`, `rainbow_hcl` and `rainbow` functions. Github–Local

Approximately 10% of men and 1% of women have some form of color blindness. The `dichromat` package provides a way of showing how a plot containing color would appear to a viewer having some form of color blindness. The package makes use of experimental data[1882] to simulate the effects of different kinds of color blindness, modifying the requested colors to appear, to normal sighted viewers, like they would to viewers having the selected kind of color blindness.

8.3.2.2 Color palette selection

Figure 8.37 shows how time varying data involving related items (in this case market share of successive versions of Android) can be displayed in a way that preferentially highlights one aspect of the data; the upper plots highlighting individual versions while the lower plots show each version's contribution to overall market share. Bold colors are effective at drawing attention to individual lines, but can be overpowering when a large area of color appears in the plot; the opposite can be the case for pastell colors.

[vii] R contains 657 built-in color names (the `colors` function lists them) and also supports hexadecimal RGB literals.

[viii] The selection process is based on theories derived from the use of color in maps,[248] which has a long history.

[ix] Red-Green-Blue (RGB) is a specification based on the display of color on computer screens; Hue-Saturation-Value (HSV) is a transformation of RGB that attempts to map to the human perceptual system and is used by some other software packages.

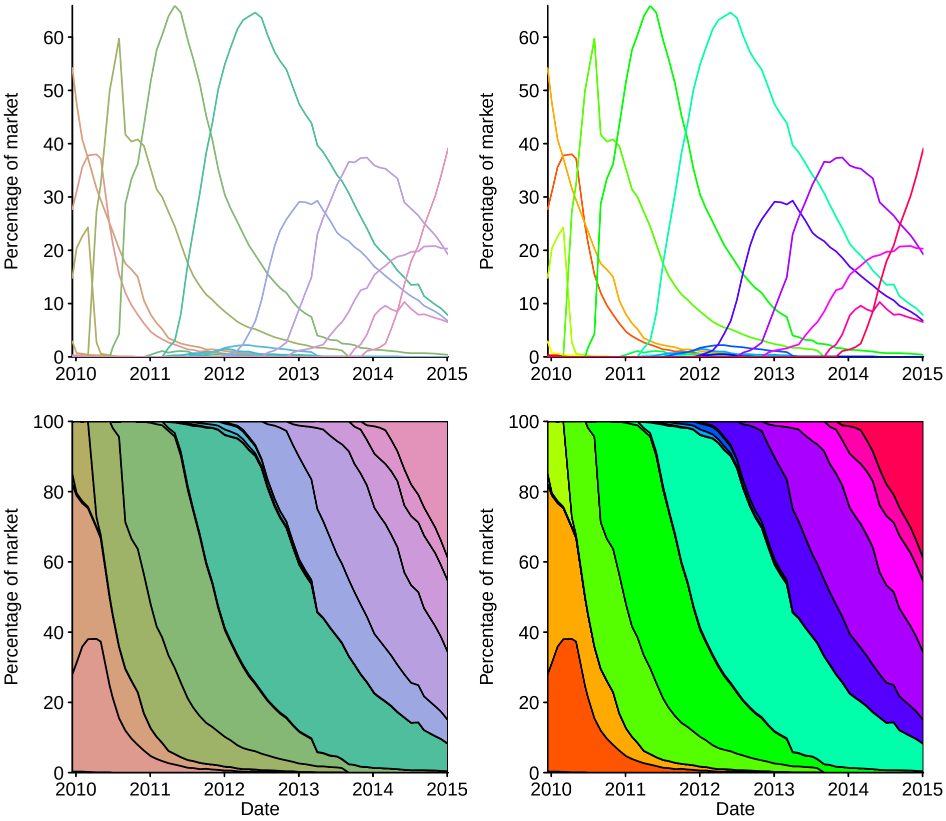


Figure 8.37: Percentage share of Android market by successive Android releases, by individual version (top) and by date (lower); pastell colors on left and bold on right. Data from Villard.[1883] Github–Local

8.3.2.3 Plot axis: what and how

The choice of plotting axis can have a dramatic impact on the visual perception of displayed data.

Linear and logarithmic are the two commonly used axis scales; square-root is common in some application domains. When the range of plotted values span several orders of magnitude, using a logarithmic axis can produce a more informative visualization; compare the use of linear and log axis in figure 8.38. Plotting values drawn from an exponential, or power law-like distribution, using a linear scale often results in many points being visually clumped together in a small area of the plot; use of a log scaled axis has the effect of spreading out these clumped values.

The `plot` function (and many other `R` plotting functions) automatically selects the minimum/maximum range of each axis, based on the range of data passed; by default 4% is added at each end of the range.

The choice of quantity plotted along each axis is driven by the relationship, between the two quantities, that the data analyst is seeking to highlight; the purpose of the plot is to visually communicate this relationship.

Care needs to be taken to ensure that artificial relationships are not generated by the choice of quantity used for one axis. An example of the wasted effort that can occur, when the relationship implied by the quantities plotted along an axis are not carefully analysed, is provided by the saga of program fault density vs. lines of code.

It was noticed that when fault density (i.e., number of faults divided by lines of code in functions) was plotted against lines of code (in functions), the distribution of points had a pattern that resembled a lopsided U. Some researchers proposed that the minimum of this U represented an optimum for the length of a function.[778]

A study by El Emam, Benlarbi, Goel, Melo, Lounis and Rai[529] showed that this U-shape was an artefact generated by the choice of quantities plotted along each axis. Plotting the ratio $\frac{F}{LOC}$ against LOC, with F constant, will produce a tilted U-shape (blue line in figure 8.39). If the number of faults grows faster than the number of lines of code (which has been found to occur for large line counts) then U-shaped curves such as the red line in figure 8.39 can occur (a growth rate was picked to illustrate one possibility; as in the following code):

```
x=1:100 ; inv.x=1/x
```

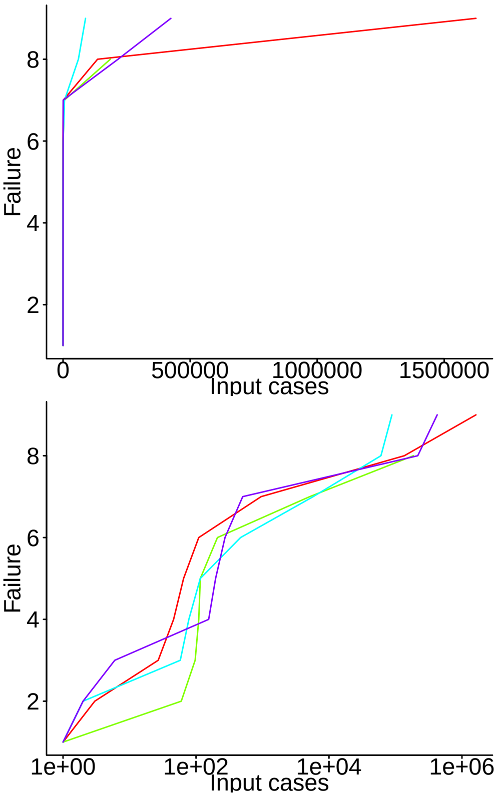


Figure 8.38: Input case on which a failure occurred, for a total of 500,000 inputs; plotted using a linear (upper) and logarithmic (lower) x-axis. Data from Dunham et al.[509] Github–Local

```
plot(x, 3*inv.x, type="l", col=pal_col[1],
                                    xlab="LOC", ylab="Faults/LOC\n")
lines(x, ((x+50)^3/5e4)*inv.x, col=pal_col[2])
```

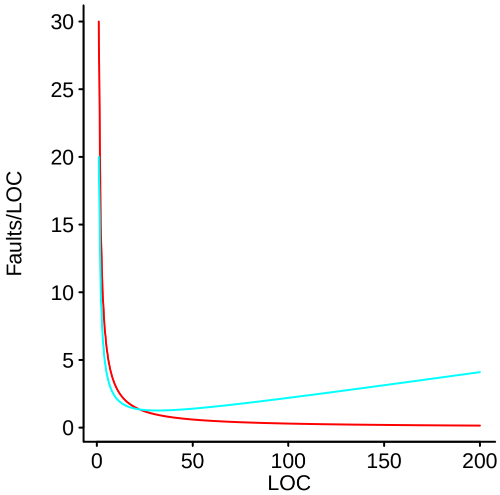


Figure 8.39: Illustration of U-shape created when y-axis values are a ratio calculated from x-axis values. Github–Local

The idea suggested by the U-shaped pattern, that there might be an optimal function length, is the result of misinterpreting the mathematical behavior of a ratio quantity plotted against one of the values used in the ratio calculation, i.e., the pattern seen in plots is an artefact of the choice of quantities chosen for each axis.

By spreading data out, a log transform of an axis can sometimes visually hide potentially useful information, rather than help reveal it. Figure 8.40 is from a study by Putnam and Myers[1517] (their Figure 8.3); in both cases the x-axis is log transformed. In the right plot the y-axis is linear, and there is a visually distinct cluster of measurements across the top; in the lower plot, where both axes are log transformed, this cluster is visually less prominent.

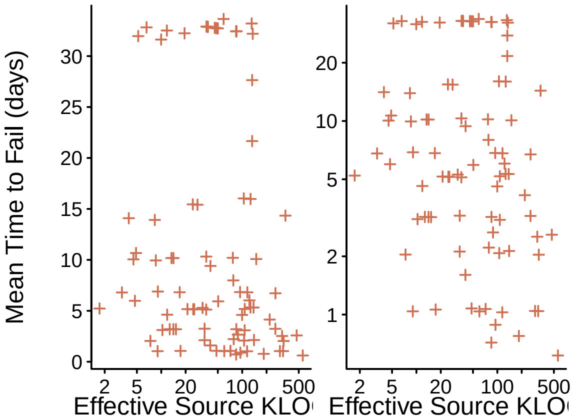


Figure 8.40: Mean time to fail for systems of various sizes (measured in lines of code); linear y-axis left, log y-axis right. Data extracted from Figure 8.3 of Putnam et al.[1517] Github–Local

8.3.3 Communicating numeric values

The output from statistical analysis can include visual plots, and a small collection of numbers. What is the best way to communicate a story involving a small collection of numbers?

The uncertainty associated with using descriptive phrases to denote probabilities is discussed in section 2.7.1 and section 6.1.4. Confidence intervals are a practical means of communicating uncertainty and are discussed in section 11.2.1. Some government organizations publish guidance on communicating uncertainty.[773]

Operation	Approximate runtime
L1 cache reference	1 ns
Branch mispredict	3 ns
L2 cache reference	4 ns
Mutex lock/unlock	17 ns
Main memory reference	100 ns
Send 2K bytes over commodity network	177 ns
Compress 1K bytes with Zippy	2,000 ns
Read 1 MB sequentially: memory	7,000 ns
SSD random read	16,000 ns
Round trip within same datacenter	500,000 ns
Read 1 MB sequentially: magnetic disk	1,000,000 ns
Seek: magnetic disk	3,000,000 ns
Send packet CA→Netherlands→CA	150,000,000 ns

Table 8.2: Numbers Everyone Should Know, circa 2016. Data from Scott.[1638]

A table of numbers covering a wide range of values (e.g., table 8.2) can be difficult to interpret quickly, unless it is something readers regularly do. An alternative representation separates out the mantissa and exponent, and combines them using area and color, allowing a visual same/different comparison to be made: as in figure 8.41.

Regression modeling (chapter 11) finds a best fit of an equation to data, according to some specified definition of the error between the equation and data. While the numeric values (often referred to as *parameters*) are the output of model building, the information being communicated is equation+parameter values, i.e., the final fitted equation should be shown.

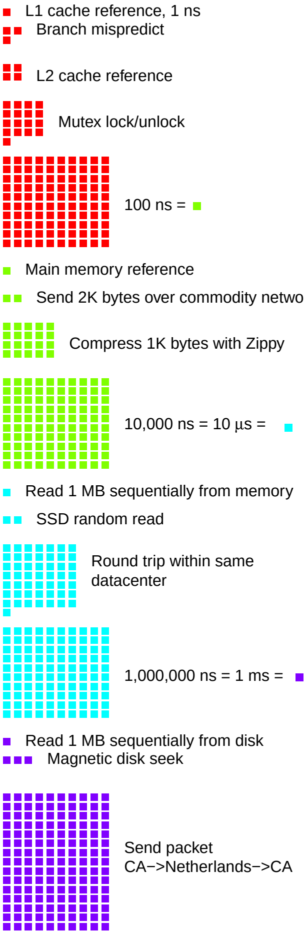


Figure 8.41: Alternative representation of numeric values in table 8.2. Data from Scott.[1638] Github–Local

Packages are available for integrating the output from R programs into the workflow of various document preparation systems, for instance, the `ascii` package provides functions for producing Asciidoc compatible output, and the `knitr` package produces LaTeX output.

Complicated equations can exhibit unexpected behavior; figure 8.42 shows the result of plotting the following set of equations, for $-4.7 \le x \le 4.7$:

$$
\begin{aligned}
y_1 &= c(1,-0.7,0.5)\sqrt{c(1.3,2,0.3)^2-x^2}-c(0.6,1.5,1.75)\\
y_2 &= \frac{0.6\sqrt{4-x^2}-1.5}{1.3 \le |x|}\\
y_3 &= c(1,-1,1,-1,-1)\sqrt{c(0.4,0.4,0.1,0.1,0.8)^2-(|x|-c(0.5,0.5,0.4,0.4,0.3))^2}-\\
&\qquad c(0.6,0.6,0.6,0.6,1.5)\\
y_4 &= \frac{c(0.5,0.5,1,0.75)\tan\left(\frac{\pi}{c(4,5,4,5)}\left(|x|-c(1.2,3,1.2,3)\right)\right)+c(-0.1,3.05,0,2.6)}{c(1.2,0.8,1.2,1) \le |x| \le c(3,3,2.7,2.7)}\\
y_5 &= \frac{1.5\sqrt{x^2+0.04}+x^2-2.4}{|x| \le 0.3}\\
y_6 &= \frac{2||x|-0.1|+2||x|-0.3|-3.1}{|x| \le 0.4}\\
y_7 &= \frac{-0.3\left(|x|-c(1.6,1,0.4)\right)^2-c(1.6,1.9,2.1)}{c(0.9,0.7,0.6) \le |x| \le c(2.6,2.3,2)}
\end{aligned}
$$

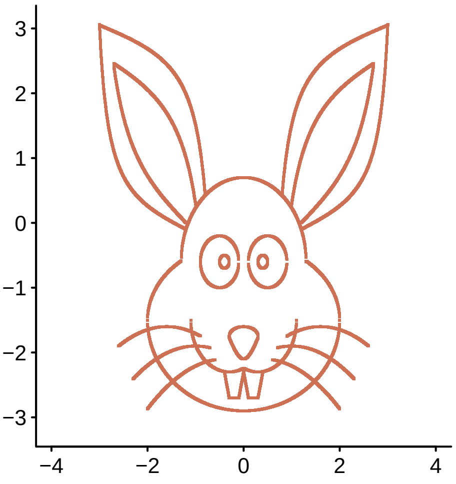


Figure 8.42: What's up doc? Perhaps, not the expected pattern in the data. Equations from White.[1926] Github–Local

8.3.4 Communicating fitted models

What is the most effective way of communicating information about a fitted model to readers?

Software developers are likely to have had lots of experience reading and interpreting equations (which are essentially a form of code). As casual users of statistical analysis, software developers will probably have to put some effort in to correctly interpreting the output produced by the `summary` function, for a fitted model; chapter 11 lists `summary` output because readers need some practice at interpreting it.

Looking at equation 11.2 (copied below), it is not necessary to search through a block of unfamiliar numbers for information about the fitted model parameters:

$$sloc = 1.139 \cdot 10^5 + 3.937 \cdot 10^2 Number\_days$$

If more information needs to be communicated, such as the uncertainty in fitted coefficients, equations enable this information to be specified at the point it applies, e.g., equation 11.3 (copied below):

$$sloc = (1.139 \cdot 10^5 \pm 1.171 \cdot 10^3) + (3.937 \cdot 10^2 \pm 4.205 \cdot 10^{-1}) Number\_days$$

The complete `summary` output (copied below) could be edited down, to remove information that is unlikely to be of interest. But trying to maintain the visual form of the `summary` output serves no useful purpose. Any additional statistical information (e.g., deviance explained) can be listed in a line of text. Github–Local

```
Call:
glm(formula = sloc ~ Number_days, data = kind_bsd)

Deviance Residuals:
   Min      1Q  Median      3Q     Max
-82990  -32136   -3609   35389   87324

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.139e+05  1.171e+03   97.24   <2e-16 ***
```

```
Number_days 3.937e+02  4.205e-01  936.33   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 1657283104)

    Null deviance: 1.4610e+15  on 4826  degrees of freedom
Residual deviance: 7.9964e+12  on 4825  degrees of freedom
AIC: 116172

Number of Fisher Scoring iterations: 2
```

As model complexity increases, readers have to invest more effort to correctly interpret equations (e.g., equation 11.6, copied below), and the number of readers willing to spend even more effort interpreting summary is likely to be less.

$$\begin{aligned} Actual = -274.8 + 1.21 Estimated + 2625 \times !D + & \\ & C_{fp}(1862 \times !D - 197.6 \times D) + \\ & C_{tp}(-2270 \times !D - 462.2 \times D) + \\ & C_{ot}(-2298 \times !D - 234.3 \times D) \end{aligned}$$

Chapter 9

Probability

9.1 Introduction

What are the chances of an event occurring?

Probability is the mathematics used to answer this question. Reasons for being interested in the estimation of probabilities include:

- betting, making an insurance decision, and all decisions and predictions involving questions about whether particular cases need to be handled,
- deciding the extent to which an event is surprising. The level of surprise might be used to decide whether something provides an opportunity or is going wrong or, when performing statistical analysis, discriminating between hypotheses.

Readers are assumed to have some basic notion of the concepts associated with probabilities, and to have encountered the idea of probability in the form of likelihood of an event occurring; classic examples involve calculating the probability of a given combination or sequence of values occurring when flipping a coin or rolling a die, e.g., two heads or rolling two sixes, or the probability of having to make N flips/rolls before some event occurs.

What is the difference between probability and statistics?

Probability makes inferences about individual events based on the characteristics of the population, while statistics makes inferences about the population based on the characteristics of a sample of the population[i].

Another way to compare the two is that probability makes use of deductive reasoning, while statistics makes use of inferential reasoning.

Probability and statistics are intertwined in that ideas and techniques from probability, about individual events, may be used when solving problems involving statistics and results about the characteristics of a population, obtained from statistical analysis, may be used to help solve problems involving probability.

People make use of various phrases to express their view of the likelihood of an event occurring (e.g., "almost impossible" and "quite possible"). Studies have found large cultural and personal differences in the numeric probabilities assigned to such phrases; see fig 6.7 and fig 2.57.

This book is data driven, and so primarily makes use of statistical analysis. The following example is a problem for which possible answers can be suggested using a probability model (data on developer behavior would provide evidence).

Say, a vendor of a static analysis tool wants to add support for detecting a newly discovered pattern of mistakes made by developers. An occurrence of this pattern, in code, is not always a mistake. What is the upper bound on the probability of generating a false positive, that keeps the likelihood of developers continuing to use the tool above some limit (say 90%)?[ii]

[i] Statistics could be defined as the study of algorithms for data analysis.

[ii] Experience shows that tool false-positives are sufficiently unpopular (they are a source of wasted effort), that a developer will stop using the tool concerned if they are encountered too often. Higher false-positive rates for Tornado warnings result in more deaths and injuries,[1688] through people ignoring the warning.

Answering this question requires knowledge of the mental model used by developers to evaluate analysis tool performance. The following are two possible mental models (both assume zero correlation between difference warning occurrences and that developers assign the same importance to all warning messages):

- an *economic* developer who tracks the benefit of processing each warning (e.g., false positive warning -1 benefit, else $+1$ benefit), starting in an initial state of zero benefit this economic developer stops processing warnings if the current sum of benefits ever goes negative.

  The Ballot theorem gives the probability that, when sequentially processing warnings, the number of true warnings is always greater than the number of false positive warnings (assuming equal weight is given to both cases, the alternative being more complex to analyse). Let C be the number of correct warnings and F the number of false positive warnings and assume $C > F$, then the probability is given by:

  $$\frac{C-F}{C+F}$$

  rewriting in terms of the probability of the two kinds of warning (i.e., $C+F=1$), we get: $C_p - F_p$

  so, for instance, when the false positive rate is 0.25 the probability of a developer processing all the warning generated by a tool is $0.75-0.25 \rightarrow 0.5$, and does not depend on the total number of warnings.

- an *instant gratification* developer who processes each warning and stops when a sequence of N consecutive false positive warnings have been encountered. This kind of thinking is analogous to that of the *hot hand in sports* (what psychologists call the clustering illusion).

  What is the probability that a sequence of N consecutive false positive warnings is not encountered?

  If the total number of warnings is k and q is the probability of a false positive occurring, then the probability of a run of N consecutive false positive warnings occurring can be calculated using the following recurrence:

  $$P(k,q,N) = P(k-1,q,N) + q^N(1-q)(1-P(k-N-1,q,N))$$

  with initial values:

  $$P(j,q,N) = 0, \text{ for } j = 0,1,\ldots,N-1$$

  $$P(j,q,N) = q^N, \text{ for } j = N$$

  Figure 9.1 shows the probability of not encountering a sequence of three (red) or four (blue) consecutive false positive warnings when processing some total number of warning messages, for various underlying false positive rates (ranging from 0.5 to 0.2).

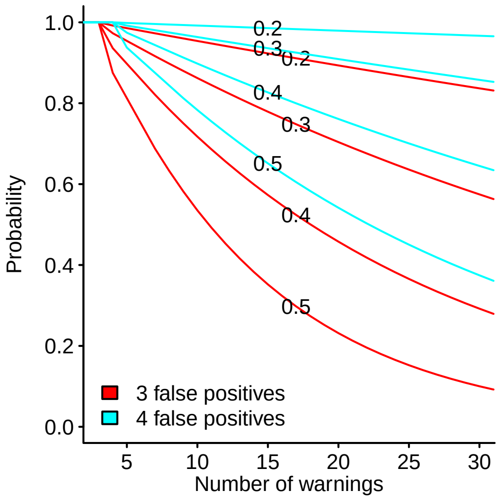


Figure 9.1: Probability that three (red) or four (blue) consecutive false positive warnings occur in some total number of warnings (false positive rate appears on line). Github–Local

When dealing with warnings involving complex constructs, a developer may be unwilling to put the effort into understanding the situation and either goes along with what the static analysis tool reports, thus underestimating the actual false positive rate, or defaults to assuming the warning is a false positive, thus overestimating the actual false positive rate.

A study by Goldberg, Roeder, Gupta and Perkins[687] investigated the ratings given to 150 jokes by 54,905 subjects. Subjects rated the jokes online, could choose whether to rate a particular joke or not, and could stop rating at any time. Figure 9.2 shows the number of subjects who rated a given number of jokes; number above 127 are somewhat erratic.

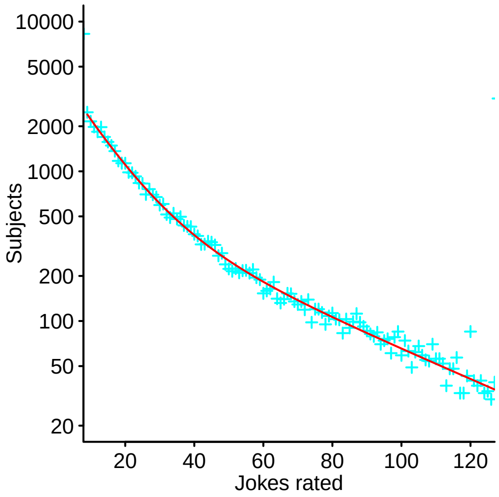


Figure 9.2: Number of subjects rating a given number of jokes, with fitted bi-exponential model. Data from Goldberg et al.[687] Github–Local

Finding an equation, or technique, to use in solving a problem involving probability requires some knowledge of the terminology used in this field. Possible phrases to try in search queries include: birth and death process, coin tossing, colored balls, combination, ergodic, event, fair games, first passage time, generating function, Markov chain, Markov process, occupancy problem, partitions, permutation, random walk, stochastic, trials and urn model.

Finding a closed form solution to an equation can be difficult, even when one exists. Sometimes the processes being studied contains so many interacting components that it is not possible to model them analytically; an alternative approach is simulation, discussed in section 12.5.

9.1.1 Useful rules of thumb

If the distribution of the values taken by some attribute, in a population, is not known, the following inequalities can be used as worst case estimates of the probability of various relationships being true. Both inequalities are distribution independent (the price of this generality is that the bounds are loose).

Markov inequality:

The Markov inequality uses the sample mean, μ, to calculate the maximum probability that X (which is required to be nonnegative) is larger than some constant. The inequality does not make any assumptions about the sample distribution:

$$P(X \geq k) \leq \frac{\mu}{k}$$

where: μ is the sample mean.

Example. If a sample has $\mu = 10$, then the probability of the sample containing a value greater than or equal to 20 (i.e., twice the mean) is: $\frac{10}{20}$.

Chebychev's inequality:

If the standard deviation, σ, of a sample is known, then Chebychev's inequality can be used to calculate a tighter bound than that given by the Markov inequality, as follows:

$$P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$$

alternatively:

$$P(|X - \mu| \geq k) \leq \frac{\sigma}{k^2}$$

Using the above example, the probability of the sample containing a value that differs from the mean by at last 10 is less than or equal to: $\frac{\sigma}{10^2}$.

Example: an analysis of the number of mutants needed to estimate test suite adequacy to within a specified error and confidence bounds.[703]

Fréchet inequalities:

Bounds on the union and disjunction of two or more probabilities are given by the Fréchet inequalities, as follows:

Logical conjunction: $\max(0, P(a_1) + P(a_2) - 1) \leq P(a_1 \wedge a_2) \leq \min(P(a_1), P(a_2))$

Logical disjunction: $\max(P(a_1), P(a_2)) \leq P(a_1 \vee a_2) \leq \min(1, P(a_1) + P(a_2))$

Correlation between three variable pairs: If the correlation between two pairs of three variables is known, say r_{12} and r_{13}, the bounds on the correlation of the remaining pair, r_{23}, is given by:

$$r_{12}r_{13} - \sqrt{(1 - r_{12}^2)(1 - r_{13}^2)} \leq r_{23} \leq r_{12}r_{13} + \sqrt{(1 - r_{12}^2)(1 - r_{13}^2)}$$

As the number of variables involved increases, the expressions become more complicated.[267]

Rule of three: Say N colored balls are drawn from a box, and the number of balls of each color counted. If there are r red balls, a reasonable estimate of the expected percentage of red balls remaining in the box is: $\frac{r}{N}$. If no green balls have been drawn, what is a reasonable estimate for the number of green balls remaining in the box?

If the fraction of green balls in the box is g, the probability of not having drawn a green ball is: $(1 - g)^N$. The 95% confidence bounds on this occurring is: $(1 - g)^N \leq 0.05$.

$$N \log(1 - g) \leq \log(0.05) \approx -Ng \leq -2.9957 \approx g \leq \frac{3}{N}$$

The non-appearance of any green balls suggests that g is very small, so: $\log(1 - g) \approx -g$.

9.1.2 Measurement scales

Mathematically, measurement values can be characterised as discrete or continuous, along with the properties of the scale used. Possible scales include the following:

- Discrete

  - *nominal scale*: each measurement value has an arbitrary number or name. Because the choice of number/name is arbitrary, no ordering relationship exists between different numbers/names. A nominal scale is not a scale in the usual sense of the word.

    Examples: the numbers on the back of footballers' shirt, or the various sales regions in which a product is sold.
  - *ordinal scale*: each measurement value is a number or name of an item, and an ordering relationship exists between the numbers/names. The distance between distinct values need not be the same. When names are assigned to entities, there may be cultural differences in the selection process. Figure 9.3 shows how words are assigned to tracts of trees having occupying various surface areas.

    Example: Classifying faults by their severity, e.g., minor, moderate, serious.

    If a minor fault is considered less important than a moderate fault, and a moderate fault is less important than a serious fault, we can deduce that a minor fault is less important than a serious fault.

    Example: The addresses of members of a C structure type is increasing, for successive members, but the difference between member addresses is not fixed because different members can have different types.

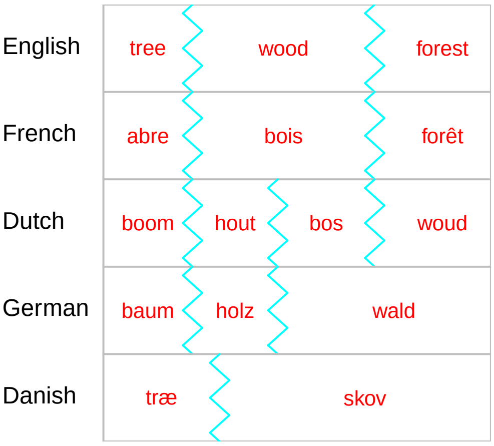


Figure 9.3: The relationship between words for tracts of trees in various languages. The interpretation given to words (boundary indicated by the zigzags) in one language may overlap that given in other languages. Adapted from DiMarco et al.[493] Github–Local

- Continuous
  - *interval scale*: each measurement is a number, a relative ordering exists, and a fixed length interval of the scale denotes the same amount of quantity being measured.

    A data point of zero does not indicate the absence of what is being measured.

    Example: the start date of some event is an interval scale. If the start date of events A, B and C are known, and the difference in start date between events A and B is the same as between events C and D, it is possible to calculate the start date of event D.

    Addition and subtraction can be applied to values on an interval scale but not multiplication or division (e.g., it makes no sense to say that the start date of event A is twice that of event C).
  - *ratio scale*: each measurement assigns a number to an item and this numeric scale preserves: the ordering of items, the size of the interval between items and the ratios between items. It differs from the interval scale in that a measurement of zero denotes the lack of the attribute being measured.

    The time difference between two events is a ratio scale.

The kinds of statistical analysis that can be legitimately performed on the values in a sample will depend on the kind of measurement scale used.

9.2 Probability distributions

Probability distributions are mathematical descriptions of the properties of values calculated by following a pattern of behavior (i.e., an algorithm). For instance, the flipping a coin pattern of behavior generates one of two results, a fixed probability of either result, with each result being independent of the previous one, and a count of the number of heads and tails has a binomial probability distribution.

If a sample of values can be fitted to a known probability distribution, then information about the pattern of behavior that generated them can be inferred from what is known about processes known to generate values having that particular distribution. For instance, given a list of pairs of numbers, if the ratio formed from each pair (i.e., $\frac{a}{a+b}$) can be fitted to a binomial distribution, there is strong evidence that the pairs are counts of a process producing one of two possible values (e.g., heads/tails, yes/no, etc.), and the probability of producing each value can be calculated from the fitted distribution.

While many probability distributions have been created,[513] only a handful of them are regularly used by analysts; R packages tend to support commonly occurring distributions, with a few packages supporting a wide range of distributions.[513]

Fitting a distribution to a sample is a step towards understanding the processes that generated the measurements, not an end in itself.

Failure to fit a known distribution may mean that more than one distribution is involved, e.g., two different coins are being used and both are biased in some way. Given enough data it is sometimes possible to obtain a reasonable fit that involves two or more distributions.

If there is reason for believing the processes being measured are driven by a known behavior, the quality of fit of the predicted probability distribution to the measured values can be compared; perhaps also against the quality of fit to other distributions.

If there is no expectation of a particular behavior, then finding an acceptable fit of some probability distribution to the measurement values is a starting point for understanding the processes that are driving the measurements observed.

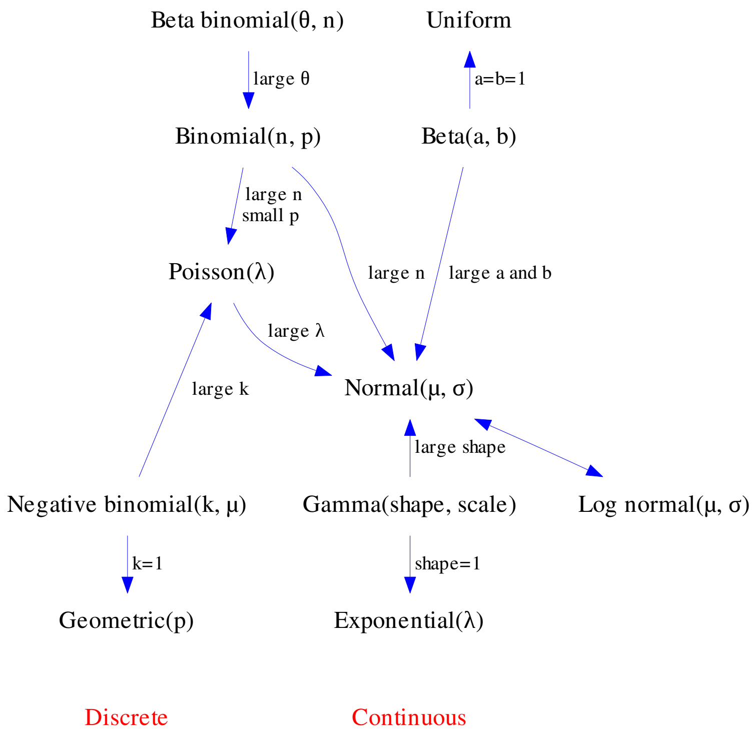


Figure 9.4: Relationships between commonly used discrete and continuous probability distributions.

Every family of probability distributions is completely characterised by a small set of numbers (often one or two) and a formula that the numbers parameterise. For instance, everything about a Normal probability distribution can be calculated by plugging values for the mean, μ, and standard deviation, σ, into the formula: $P(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-(x-\mu)^2/2\sigma^2}$ (this formula is often abbreviated as $N(\mu,\sigma)$). Fitting data to a Normal distribution involves finding appropriate values for μ and σ.

In practice, a few probability distributions are encountered much more often than others. One of these common cases will often fit reasonably well to a wide spectrum of commonly encountered samples, and unless there are theoretical reasons for expecting a less commonly encountered distributions, there is nothing to be gained by searching through all known distributions to find the one that best fits a sample.

Some characteristics of sample values, that may or may not correspond to a known probability distributions include:

- mean value (sometimes called the *arithmetic mean*, *central tendency* or *location value*, the last two terms may also be used to refer to the median): many distributions have a finite mean (examples that don't include power laws with an exponent greater than or equal to -1 and the Cauchy distribution),
- scale parameter, *variance* (*standard deviation* is the square-root of variance): how spread out the distribution is; a few distributions do not have a finite variance, e.g., power laws with an exponent between zero and -2,
- the extent to which the distribution is symmetrical/asymmetrical about its mean, the *skew* of a distribution is a measure of how asymmetrical it is; a symmetric distribution has a skew of zero, while a positive skew has a tail pointing towards larger positive values, and a negative skew has a tail pointing towards negative values,
- where most of a distribution's density resides, e.g., around the mean or in the tails. The *kurtosis* of a distribution is a measure of how spiky the distribution is; possibilities include tall and slim (known as *leptokurtic*; slender-curved), short and flat (known as *platykurtic*) or medium-curved (known as *mesokurtic*; the Normal distribution has a Kurtosis of three),

- number of distinct peaks, known as the *modality* of a distribution; a distribution with one distinct peak is said to be unimodal, two distinct peaks bimodal (such as measurements from two different distributions, e.g., height of men/women).

The moments package contains functions for calculating skewness, kurtosis and moment related attributes of a numeric vector.

Probability distributions can be divided into discrete and continuous distributions, with discrete distributions only being defined at specific points (usually integer values). In R, functions that involve discrete distributions usually require integer values while functions involving continuous distributions take floating-point values.

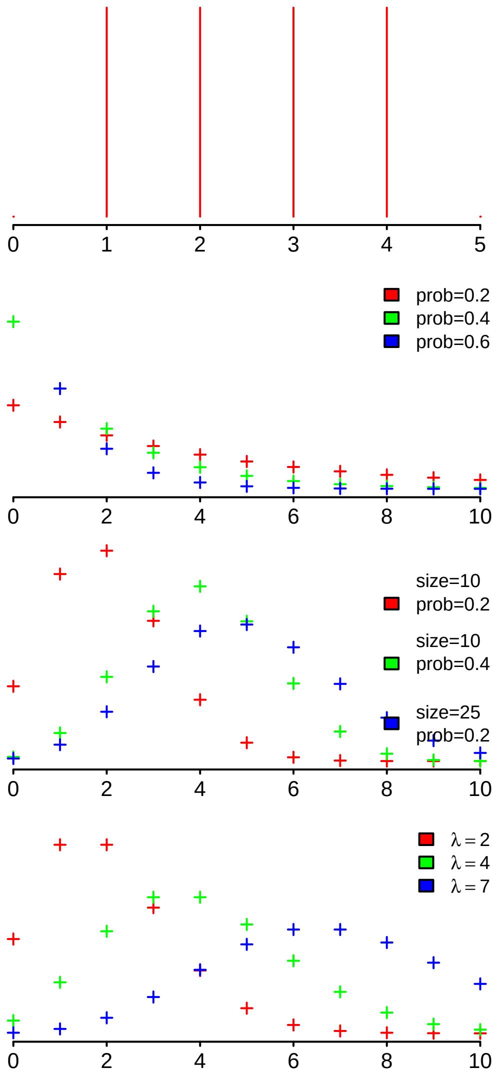


Figure 9.5: Shapes of commonly encountered discrete probability distributions (upper to lower: Uniform, Geometric, Binomial and Poisson). Github–Local

There are various ways of representing a probability distribution, and the following are often encountered:

- density function: for discrete distributions (see figure 9.5) this can be viewed as the probability that x will have a given *value*, $P(x = value)$; for continuous distributions (see figure 9.7) the probability of any particular value occurring is zero, however there is a finite probability of a measurement returning a value within a specified interval,
- cumulative density function: the probability that x will be less than or equal to a given *value*, $P(x < value)$, see figure 9.6,
- equation: an equation for the probability distribution. For the majority of people (including your author), this is little more than eye candy, e.g., the equation, $\frac{\lambda^k e^{-\lambda}}{k!}$, is very difficult to visualize and is only of use to developers wanting to implement the Poisson distribution.

Discrete distributions: commonly encountered discrete distributions include the following (see figure 9.5):

- *Binomial distribution*: for a random variable X,

  1. the process involves a sequence of independent trials,
  2. each trial produces two possible outcomes, e.g., heads/tails,
  3. the probability of either outcome (p, say, for heads) does not change,

     – X counts the number of success (where success might be defined as a head occurring) in n fixed trials.
     The Binomial distribution is completely described by two parameters: $B(n, p)$, This process is sometimes described using the analogy of, drawing n objects from a pool containing a finite number of two kinds of object, where the object is placed back in the pool after it has been drawn (this kind of draw is said to be: *with replacement*). The Hypergeometric distribution is the result, if objects are not returned to the pool once they are drawn (this kind of draw is said to be: *without replacement*).
     A distribution supporting more than two discrete values is known as a *Multinomial distribution* (again with a fixed probability of each value occurring). The XNomial package provides support for multinomial distributions,

- *Negative Binomial distribution*: this has the same three requirements as the Binomial distribution, but differs in what is counted,

  – X counts the number of trials up to and including the k^{th} success (where success might be defined as a head occurring after a continuous sequence of tails).
  A process that produces values having a Negative Binomial distribution is randomly drawing from a mixture of Poisson distributions, where the mean of the mixture of Poisson distributions has a Gamma distribution,
  This distribution is a generalised version of the Geometric distribution (which is based on the probability of observing the first success on the n^{th} trial).

- *Poisson distribution*: for a random variable X,

  1. the process involves independent events,
  2. only one event can occur at any time,

     – X counts the number of events that occur within a specified time.
     The Poisson distribution is complete described by one parameter (λ, the distribution mean): $P(\lambda)$,

The sum of two independent Poisson distributions $P(\lambda_1)$ and $P(\lambda_2)$ is the Poisson distribution: $P(\lambda_1+\lambda_2)$.
The Binomial and Poisson distributions are related in that as $n \to \infty$ and $p \to 0$, then $B(n,p) \to P(np)$, i.e., The Poisson distributions is a limit case of a Binomial distribution having a very low probability of success over a long period.

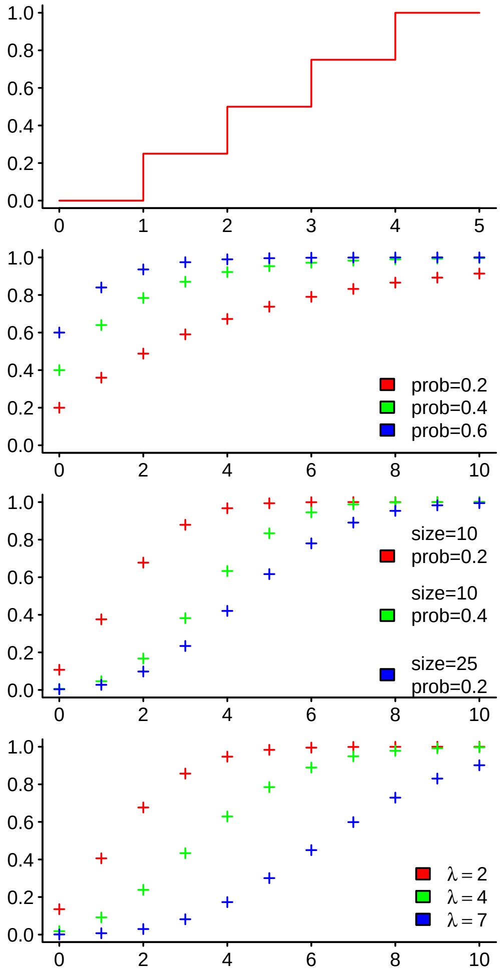


Figure 9.6: Cumulative density plots of the discrete probability distributions in figure 9.5. Github–Local

Continuous distributions: commonly encountered continuous distributions include the following (in all but one case, the generating process clusters the values around a single peak; see figure 9.7):

- *Uniform distribution*: all values between the lower and upper bounds of the interval have an equal probability of occurring, i.e., no value is more likely to occur than any other. For discrete values between 1 and n the probability of any value occurring is $\frac{1}{n}$.

  One process that generates a uniform distribution is a random number generator, such as calling R's `runif` function.

- *Normal distribution*: can be generated by adding together contributions from many independent processes; a consequence of the Central limit theorem. This distribution crops up with great regularity, it has a mathematical form that is easier to manipulate analytically, than many other distributions, resulting in it being widely used before computers reduced the need for analytic solutions to equations. This distribution is described by its mean and variance.

  While the Normal distribution is the result of adding contributions from many independent processes, it is not true to say that adding contributions from many different kinds of processes will result in this distribution (similarly, for multiplicative contributions and a lognormal distribution). For instance, given the right conditions, adding values drawn from many different Poisson distributions can result in a Negative Binomial distribution, a Geometric distribution or other distributions,[964]

- *Lognormal distribution*: the logarithm of a Normal distribution, which can be thought of as being generated by multiplying together the sum of contributions from many independent processes;[1282] samples drawn from a Lognormal distribution can produce a straight line, over some of their range, when plotted using log-log axis,

- *Exponential distribution*: generated by a memoryless process, e.g., the waiting time for an event to occur is independent of the amount of time that has passed since the last event. This is the continuous form of the Geometric distribution, and like it, is described by a single parameter.

  Over some of its range the exponential distribution is visually similar to a power law, which has led researchers to incorrectly claim that their sample fits a power law (a fashionable distribution to have one's sample following; see section). Power laws, and associated scale-free networks are rare in many application domains, but common in a few technological networks.[255]

  The sum of a reasonably large number of independent exponential distributions has an Erlang distribution, e.g., the interval between incoming calls to a telephone exchange, where the interval between calls from any individual have an exponential distribution,

- *Beta distribution*: applies to processes where the explanatory variable is restricted to a finite interval, e.g., the interval zero to one. This distribution is defined by two, non-negative, shape parameters.

- *Gamma distribution* (Γ is the Greek uppercase Gamma, the symbol often used to denote the Gamma function, is the lowercase version, γ): used to describe waiting times, e.g., `Gamma(shape=3, scale=2)` is the distribution of the expected waiting time (in some units) for three events to occur, given that the average waiting time is 2 time units (yes, the `Gamma` function differs from most other distribution names in the base system by starting with an uppercase letter).

  When `shape=1`, the Gamma distribution reduces to the Exponential distribution.

  The Gamma distribution is the continuous equivalent of the Negative binomial distribution.

- *Chi-squared distribution* (sometimes written using χ, the Greek lowercase letter of that name): is more often encountered in the mathematical analysis of statistics, than as a distribution of a sample. A random variable has a chi-squared distribution, with d degrees of freedom, if it is produced by a process which generates the values: $Z_1^2 + Z_2^2 + \cdots + Z_d^2$, where Z_i are independent random variables having a Normal distribution.

  The chi-squared distribution is a special case of the Gamma distribution.

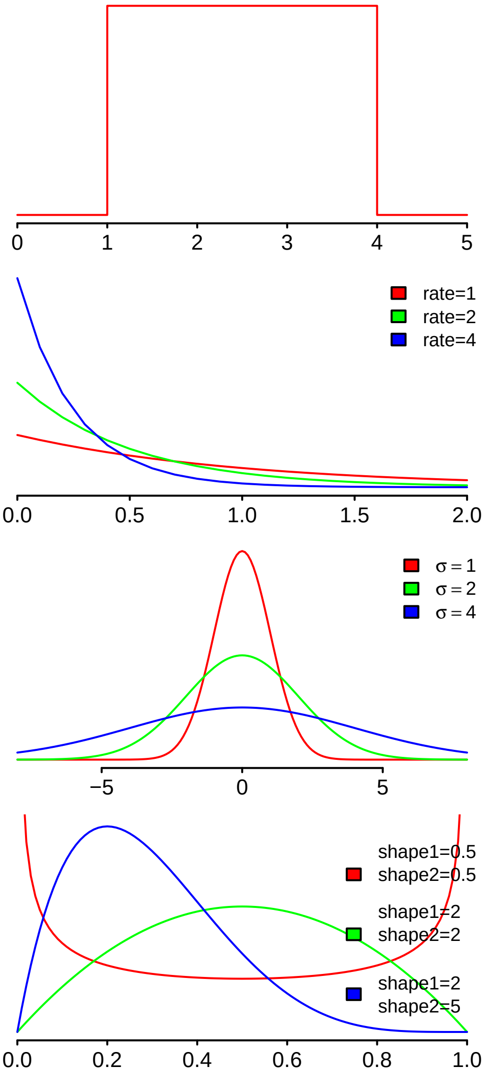


Figure 9.7: Commonly encountered continuous probability distributions (upper to lower: Uniform, Exponential, Normal, beta). Github–Local

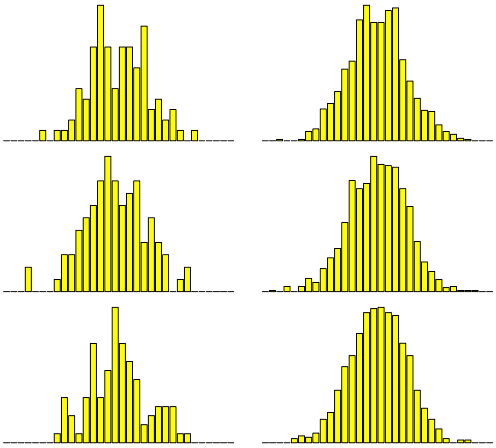


Figure 9.8: Samples of randomly selected values drawn from the same normal distribution (left: 100 points in each sample, right 1,000 points in each sample). Github–Local

- *Weibull distribution*: this distribution drops out as the solution to various problems in hardware reliability, e.g., time to failure, and is often used as the hazard function in survival analysis. The Exponential and Rayleigh distributions are special cases of the Weibull distribution,
- *Cauchy distribution*: this distribution is more famous for its unusual characteristics, e.g., having an undefined mean and variance (because of its very fat tail), than through its uses. The density function for the average of two random variables each having a Cauchy density is a random variable with a Cauchy density; this self mapping is unique to the Cauchy distribution. One consequence is that, if the error in a measurement has a Cauchy density, then the average of many measurements will not be more accurate than the individual measurements.

9.2.1 Are two sample drawn from the same distribution?

As always, visualization is a useful first step in judging whether two samples might be drawn from the same distribution. However, be warned, small datasets can produce visualizations showing little resemblance to the distributions from which they were drawn; as can be seen from figure 9.8, where all the samples are drawn from the same Normal distribution.

A study by Veytsman and Akhmadeeva[1876] measured subject reading rate, in words per minute, for text printed using a Serif or Sans Serif font. Words per minute is a discrete distribution and subject performance is likely cluster around similar values, i.e., there will be duplicates. Figure 9.9 shows a density plot of the normalised data.

The various comparison methods are based on some measure of difference between the *shape* of the sample distributions. The following tests are based on comparing the edf (empirical distribution function) of the samples.

- The Anderson-Darling test is based on the largest difference between the edf of the two distributions, it uses weights to ensure that the tails of the distribution have as much influence as other parts of the distribution; it is possible to use this test to compare more than two distributions. While the Kolmogorov-Smirnov test is often encountered, it has been found to be less sensitive than the Anderson-Darling test[1754] because it primarily detects differences in the main body of the distribution, rather than over the complete range of values.

  The `ad.test` function in the `kSamples` package implements the Anderson-Darling test for two or more samples.

  The `ks.test` function, part of the base system, implements the Kolmogorov-Smirnov test; other implementation include the `ks.test` function in the `dgof` package whose interface is the same but includes support for discrete distributions.

  Samples drawn from a continuous distribution are very unlikely to contain identical values, and many implementations warn if a sample contains duplicate values.
- The Cramér-von Mises test is based on summing (the square of) differences between edfs, rather than using a single maximum value, and can be more powerful against a large class of alternative hypothesis.[72]

  The `cvm.test` function in the `dgof` package implements the Cramér-von Mises test.

The bootstrap can be used to estimate the probability of two sample distributions differing by the amount reported by the statistical test used.

The choice of statistical test depends on whether differences over the range of values in the samples are of interest, whether tail values are uninteresting (perhaps because there are few measurements in the tail, and so what is there is noisey), or the amount of difference between sample distributions is the primary differentiator.

Comparison of samples drawn from discrete distributions is provided by the `WRS` package (on Github), which implements a version of the Kolmogorov-Smirnov test (the `ks` function) that supports discrete data, and also the `bmpmul` function that uses the Brunner-Munzel test (also see the `ks.test` function in the `dgof` package).

The following code performs various tests that check whether the two sample are likely to have been drawn from the same population (see Github–group-compare/tb104veytsman-dist.R):

```
library("dgof")
library("kSamples")
library("WRS")

# From WRS
ks(serif$Standard_WPM, sansserif$Standard_WPM)
# In fact unscaled measurements give the same result, i.e., not different
ks(serif$WordsPerMinute, sansserif$WordsPerMinute)

dgof::ks.test(serif$Standard_WPM, ecdf(sansserif$Standard_WPM))

# From base system
ks.test(serif$Standard_WPM, sansserif$Standard_WPM)

# Only applicable to continuous distributions
ad.test(serif$Standard_WPM, sansserif$Standard_WPM)
```

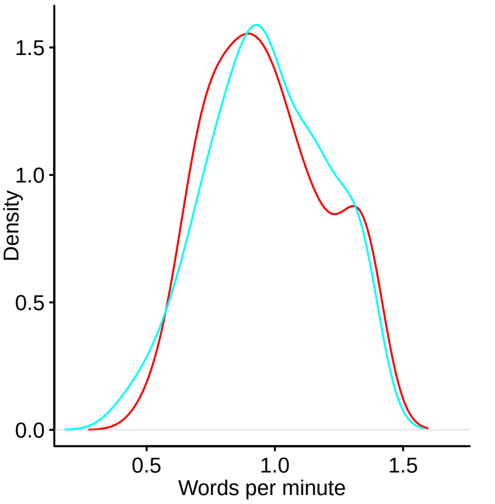


Figure 9.9: Reading rate for text printed using a serif (blue) and sans-serif (red) font, data has been normalised and displayed as a density. Data from Veytsman et al.[1876] Github–Local

The hypothesis that the samples plotted in figure 9.9 are drawn from populations having different distributions is rejected.

Note that many measurement points may be needed to reliably detect a difference in distributions, when one exists. For instance, when one sample is drawn from an Exponential distribution and the other from a Normal distribution, two samples of 150 points are needed to obtain a 95% confidence level, using `ad.test`, that the samples are drawn from different distributions (550 points are needed when the samples are drawn from Normal and Uniform distributions); see Github–group-compare/ad-check.R.

For some analysts, testing whether a sample is drawn from a Normal distribution is a common activity (techniques that are practical to perform manually often require that samples be drawn from this distribution[iii]).

The result of testing whether a small sample is drawn from a Normal distribution has a high degree of uncertainty. The points in figure 9.10 was obtained by testing samples, all drawn from the same distribution (e.g., via a call to `rexp`), using the `shapiro.test` function (replicated 1,000 times for each sample size). The y-axis shows the probability of the Shairo-Wilk test detecting that the sample values are not drawn from a Normal distribution (p-value < 0.05; when the values have been drawn from another distribution); for the case when the values are drawn from a Normal distribution (e.g., a call to `rnorm`) the y-axis gives the probability of this fact not being detected.

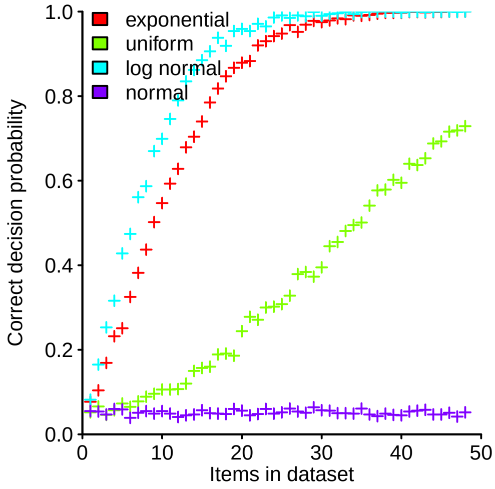


Figure 9.10: Probability, with p-value < 0.05, that `shapiro.test` correctly reports that samples drawn from various distributions are not drawn from a Normal distribution, and probability of an incorrect report when the sample is drawn from a Normal distribution; 1,000 replications for each sample size. Github–Local

There is no guarantee that the values in a sample have a distribution that even closely resembles any known probability distribution.

A study by Berger, She, Czarnecki and Wąsowski[176] investigated the use of feature macros used in the configuration of software product lines. Figure 9.11 shows the number of conditionally compiled sections of source code that were dependent on a given number of feature macros.

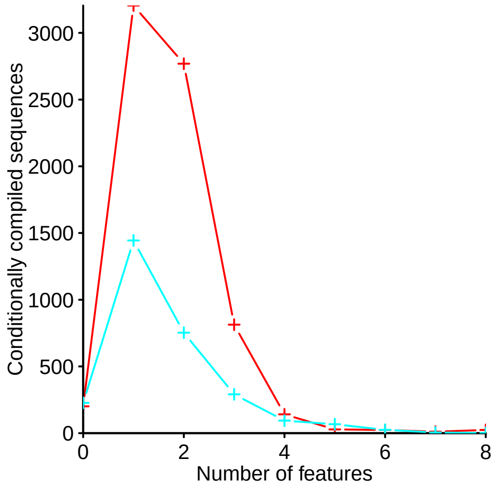


Figure 9.11: Number of conditionally compiled code sequences dependent on a given number of feature macros (red overwritten by blue: Linux, blue: FreeBSD). Data from Berger et al.[176] Github–Local

A Cullen and Frey graph shows that the characteristics of neither sample are close to matching any common discrete distributions. A Kolmogorov-Smirnov test considers them to be sufficiently different, that they are likely to have been drawn from different distributions (see Github–group-compare/cond-compile/2010-berger.R).

Samples may appear to have a similar shape, but have different mean values. Technically, samples with different mean values (or standard deviations) are considered to be drawn from different distributions. There may be theoretical reasons for believing that samples have been generated by the same processes and normalizing mean values (or even variance) enables the shape of the sample distributions to be compared.

A study by Zhu, Whitehead, Sadowski and Song[2005] counted the number of various kinds of statements in a corpus of C, C++ and Java programs (approximately 100 programs, around 10 million lines, for each language). Figure 9.12 shows the distribution of occurrence (expressed as a density on the y-axis) of various statements (expressed as a percentage on the x-axis), over the programs measured; a different color for each language, figure out which is which, before looking at the code.

[iii] Readers of this book learn techniques that don't have this precondition

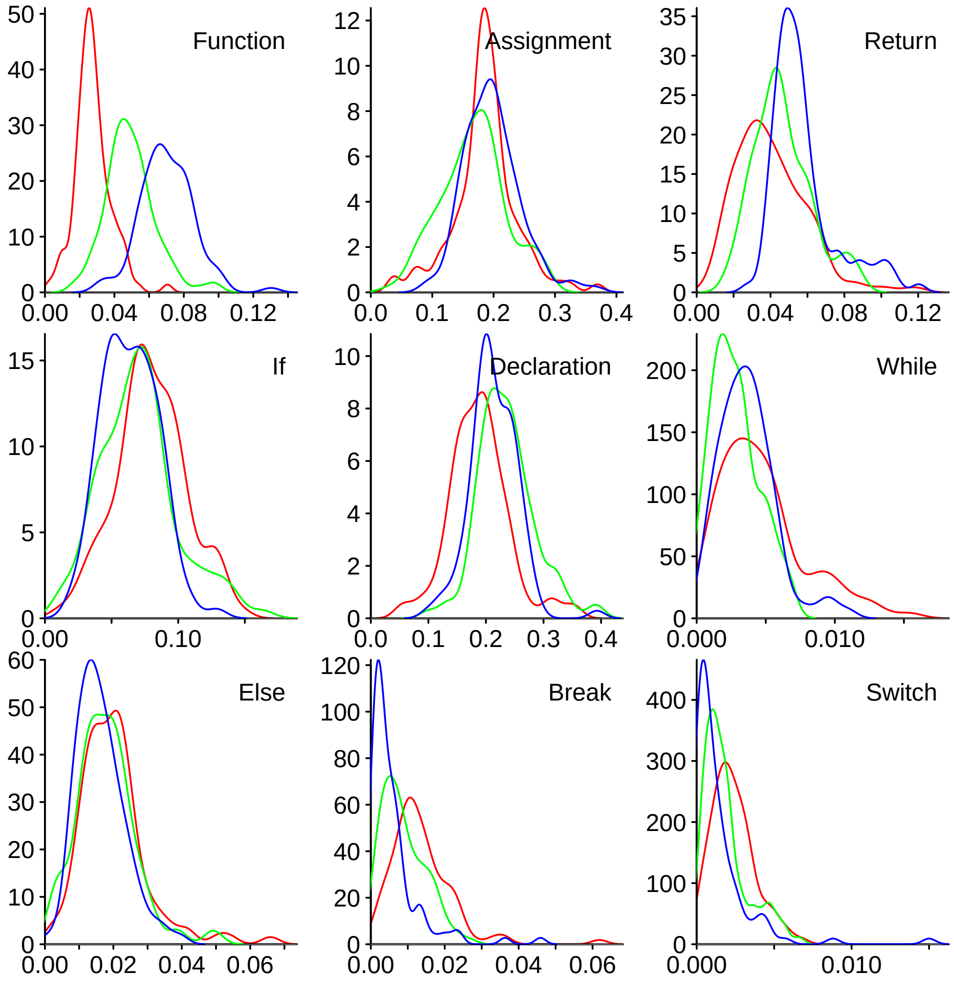


Figure 9.12: Percentage occurrence of statements (x-axis) for each of 100 or so C, C++ and Java programs (colored lines, figure it out or look at the code), plotted as a density on the y-axis. Data from Zhu et al.[2005] Github–Local

Differences in the probability of various kinds of statements being used, over a sample of programs written in various languages, is evidence that language has an impact on what code gets written (either because particular kinds of applications are written using a given language, particular algorithm selection is influenced by language, or the impact of differences in language semantics).

Might two or more of the languages measured be said to have the same distribution of `if-statement` and/or `assignment-statement` usage? The interactions between different statements makes the analysis non-trivial.

The takeaway from this section is that for small sample sizes, distribution comparison produces unreliable answers, and for large samples comparison may be complicated.

Comparison of particular characteristics of sample distributions, e.g., sample means, is discussed in section 10.5.

9.3 Fitting a probability distribution to a sample

Given a sample of values, which of the known, supported by `R`,[513] probability distributions is the best fit?

There is no universal best-test statistic, for goodness-of-fit of a sample to a probability distribution. The performance of the available tests depends on the (unknown) distribution from which the sample was drawn.[1746]

The Normal distribution is often the default answer given, when people are asked about the distribution of a sample. There are several reasons for this, including: historically many techniques designed to be performed by a human calculator were derived from theory that assumed normally distributed data (which often appeared to work reasonably well, when the data only approximated a Normal distribution), along with a misunderstanding of what the Central Limit theorem is about, driving a belief that a complex process provides the mixing needed to produce a Normal distribution.

As always, knowledge of the processes driving the production of measured values can be very useful. For instance, measurements of arrival times that are driven by a Poisson process will result in inter-arrival times that are exponentially distributed, values created via the multiplicative effect of many contributions may have a Lognormal distribution, and a preferential attachment process often results in links or what they link-to following a power law.

If there is no theoretical justification for a particular distribution, limiting the selection process to those distributions having some degree of name recognition is likely to make the one chosen an easier sell to readers. For instance, the Delaporte distribution[iv] might happen to fit a particular sample slightly better than the Negative Binomial distribution, but its lack of name recognition means that extra effort will have to be invested, justifying its use.

A study by van der Meulen[1857] posted the $3n+1$ problem on a programming competition website: 95,497 solutions were submitted and van der Meulen kindly sent me a copy of these solutions (11,674 solutions were written in Pascal, the rest in C). The $3n+1$ problem is: write a program that takes a list of integers and outputs the *length* of each value, where length is the number of iterations of the following algorithm:

```
for input integer ++pass:[n]++;
   while (n != 1)
      n = (is_even(n) ? n/2 : n*3+1);
```

Which distribution is a good approximation, to the number of lines of code contained in the programs submitted as answers to this problem?

The first step of visualizing the sample provides basic information about the shape of the distribution, e.g., decreasing/increasing, single/multiple peak, symmetric/skewed or appearing to be nothing but random noise (see figure 9.14).

A method of narrowing down the list of possible distributions, is to plot a Cullen and Frey graph. The `descdist` function, in the `fitdistrplus` package, plots this graph and returns some descriptive distribution characteristics of the values (mean, median, sd, skewness and kurtosis). Skew and kurtosis are not reliable estimators and `descdist` includes an option to create and test bootstrap samples.

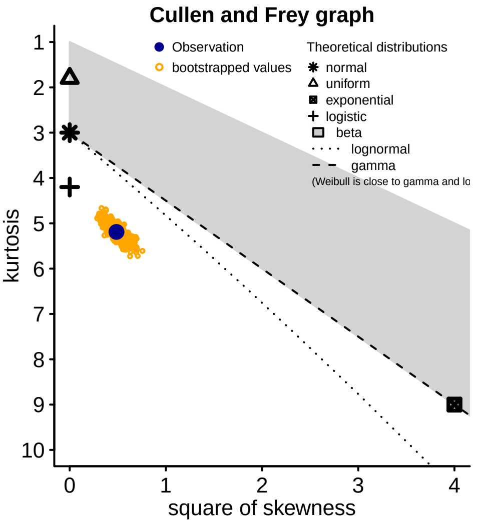


Figure 9.13: A Cullen and Frey graph for the $3n+1$ program length data. Data kindly provided by van der Meulen.[1857] Github–Local

The blue circle and yellow points in figure 9.13 denote the sample and various bootstrapped results for the $3n+1$ program lengths, assuming a continuous distribution (the average number of lines is large enough that the difference between discrete/continuous is likely to be small). The sample does not overlay any of the grey lines/areas on the plot that denote commonly occurring distributions. The code is:

```
library(fitdistrplus)

# Default is to check continuous distributions
# dummy=descdist(li, discrete=TRUE, boot=500)
dummy=descdist(li, boot=500)
```

The `fitdist` function[v] in the `fitdistrplus` package can be used to fit a distribution to the data, i.e., find values of the specified distribution's parameters, such as mean and variance, that minimise some measure of goodness-of-fit (the AIC of the fit is returned). The `gamlss` package supports a wider range of distributions (see the help information for the `gamlss.family` function) that `fitdist` can use to fit data.

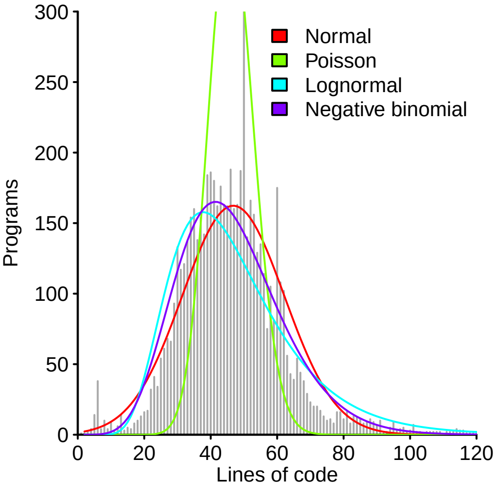


Figure 9.14: Number of 3n+1 programs containing a given number of lines, with four distributions fitted to this data. Data kindly provided by van der Meulen.[1857] Github–Local

Figure 9.14 shows fits for the Normal, Poisson, Lognormal and Negative binomial distributions.

```
library(fitdistrplus)

tp=fitdist(li, distr="pois"); tnb=fitdist(li, distr="nbinom")
tn=fitdist(li, distr="norm"); tln=fitdist(li, distr="lnorm")

# gofstat is a way of getting all the values used for plotting
theo_vals=gofstat(list(tn, tp, tln, tnb), chisqbreaks=1:120,
                        fitnames=c("Poisson", "Negative binomial",
```

[iv] A compound distribution derived from a Poisson distribution whose mean has a shifted Gamma distribution.

[v] The `MASS` package contains the `fitdistr` function and the `gamlss` package contains the `fitDist` function, both of which fit distributions to data.

```
                                "Normal", "Lognormal"))

plot_distrib=function(dist_num)
{
lines(theo_vals$chisqbreaks, head(theo_vals$chisqtable[, 1+dist_num], -1),
                col=pal_col[dist_num])
}

plot(theo_vals$chisqbreaks, head(theo_vals$chisqtable[, 1], -1), type="h",
              xlab="Program length", ylab="Number of programs\n")
plot_distrib(1); plot_distrib(2)
plot_distrib(3); plot_distrib(4)
```

The large spike at 50 lines might be caused by solutions all doing the same thing, but with different statement orderings, e.g., multiple submissions derived from a common solution.

Based on minimizing AIC, the Normal distribution is the best fit, with the Negative binomial distribution a close second. Should either distribution be chosen as the best fitting, or is it worthwhile attempting to fit other distributions? The answer depends on what the fitted distribution will be used for, e.g., making predictions or building models. Jumping to any conclusions based on one data-point (i.e., set of length measurements for one problem) is always problematic.

9.3.1 Zero-truncated and zero-inflated distributions

Some distributions only make use of non-negative values, they start at zero, e.g., the Poisson distribution. While zero is a common lower bound for measurement values, other lower bounds occur, e.g., the number of minutes to complete a task (the zero time tasks, that are never started, are not measured).

It is possible to adjust the equations that describe zero-based distributions, to have a non-zero lower bound. Rebasing a distribution to start at one (rather than zero) is the common case and after such an adjustment the distribution is said to be *zero-truncated*, e.g., *zero-truncated Poisson distribution*.

The gamlss.tr package contains functions that support the creation of zero-truncated (or truncation to the right or left of any value) distribution functions. The following code creates a set of functions relating to the zero-truncated type II Negative binomial distribution; the name of the created function is NBIItr and like other distribution functions in R, the associated density, distribution, quantile and random functions are obtained by prefixing the letters d, p, q and r, respectively, to NBIItr:

```
library(gamlss)
library(gamlss.tr)

gen.trun(par=0, family=NBII) # Bring various functions into existence
```

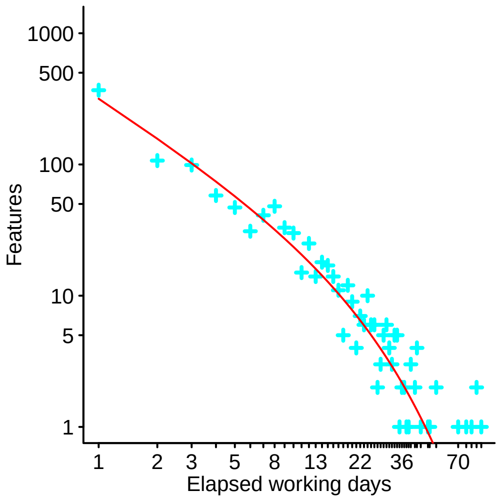


Figure 9.15: A zero-truncated Negative Binomial distribution fitted to the number of features whose implementation took a given number of elapsed workdays; first 650 days used. Data kindly provided by 7digital.[1] Github–Local

The 7Digital data[1] (discussed in more detail in section 5.4.6) contains information on 3,238 features implemented between April 2009 and July 2012; the information consists of three dates (Prioritised/Start Development/Done), from which a non-zero duration can be calculated.

The Cullen and Frey graph suggests a negative binomial distribution might be a good fit.

The functions returned by gen.trun do not have a form that can be used in calls to the fitdist function. The gamlss function in the gamlss.tr package has a special form for handling these created functions, as shown in the following code (where day_list contains the list of values and NBIItr was created by an earlier call to gen.trun). The following code was used to produce figure 9.15:

```
library(gamlss)
library(gamlss.tr)

g.NBIItr=gamlss(day.list ~ 1, family=NBIItr)

NBII.mu=exp(coef(g.NBIItr, "mu"))       # get mean coefficient
NBII.sigma=exp(coef(g.NBIItr, "sigma")) # standard deviation

plot(table(day.list), log="xy", type="p", col=point_col,
                xlab="Elapsed working days", ylab="Features\n")

lines(dNBIItr(1:93, mu=NBII.mu, sigma=NBII.sigma)*length(day.list), col="red")
```

One process generating values having a Negative binomial distribution is based on a mixture of Poisson distributions, whose means have a Gamma distribution. It is possible to generate other distributions by combining a mixture of Poisson distributions, are any of these a better fit to the data? The Delaporte distribution sometimes fits slightly better and sometimes slightly worse; the difference is not large enough to warrant switching from a relatively well-known distribution, to one that is rarely covered in text books or supported in software; if data from other projects is best fitted by a Delaporte distribution, then it may be worthwhile spending time analysing how this distribution might be a better model of project scheduling.

If the processes generating these values can be modeled by a mixture of Poisson distributions, it is unlikely that a single subprocess is responsible for a large percentage of the quantity measured, many subprocesses are involved.

Sometimes count data contain many more zero values than are expected, from the distribution that the generating process is believed to follow. Two kinds of behavior that can cause an excess of zeroes to appear in the measurements are:

- a process that generates zeroes, and a process that generates non-negative values; this situation can be modeled by what is known as *zero-inflated model*. The gamlss package supports zero-inflated distributions,
- the measurements involve two processes, one where the values are zero or non-zero, and the other where values are always non-zero (i.e, zero-truncated); this situation can be modeled by what is known as a *hurdle model* (the hurdle that has to be got over is moving from zero to non-zero). The gamlss package supports what it calls *zero altered* (or *zero adjusted*) distributions, while the pscl package uses the term hurdle.

Your author's search for software engineering measurements containing an excess of zeroes located a few that appeared to contain an excess, but none could be fitted by the models discussed above (see Github–probability/bolz_data_struct_racket.R for an example).

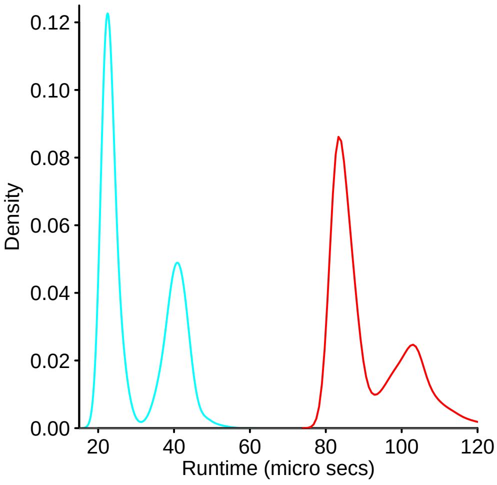


Figure 9.16: Density plot of MPI micro-benchmark runtime performance for calls to MPI_Allreduce with 1,000 Bytes (left curve) and to MPI_Scan with 10,000 Bytes (right curve). Data kindly supplied by Hunold.[866] Github–Local

9.3.2 Mixtures of distributions

Sometimes sample measurements are generated by two or more distinct processes, resulting in values that appear to be drawn from two or more distinct distributions, e.g., a plot shows multiple peaks. A model built using a mixture, or weighted sum, of distributions is known as a *finite mixture model* or just a *mixture model*; a continuous mixture of distributions is known as a *compounded distribution* (the Negative Binomial distribution is a compounded distribution).

The mixtools and rebmix packages contain functions for fitting samples drawn from two or more of the same kind of distribution family, e.g., multiple Normal distributions. The two packages differ in the structure of their API, e.g., one having many functions, and the other having one main function taking many arguments (neither would win a prize for user interface design).

A study by Hunold, Carpen-Amarie and Träff[866] investigated the impact of external factors on the performance of an MPI micro-benchmark. Figure 9.16 shows the runtime variation of two different MPI calls, with each having two distinct peaks. The two peaks in the left curve appear to be symmetrical and perhaps a mixture of two Normal distributions is a good fit. Figure 9.17 shows the two distributions fitted by a call to the normalmixEM function (in the mixtools package), along with a histogram (all produced by the same call to the plot function provided by the package).

```
library("mixtools")

scan_dist=normalmixEM(fig1_Allreduce$time)

plot(scan_dist, whichplots=2, main2="", col2=pal_col,
        xlab2="Time (micro secs)", ylab2="Density\n")
```

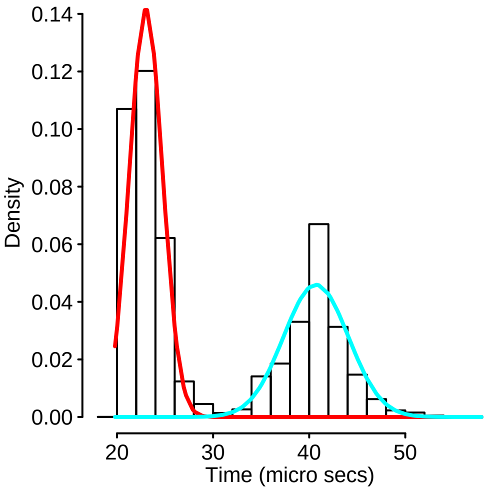


Figure 9.17: Mixture model fitted by the normalmixEM function to the performance data from calls to MPI_Allreduce. Data kindly supplied by Hunold.[866] Github–Local

A call to summary returns the parameters of the fitted model; the first row (prefixed by lambda) is the fraction contributed by each distribution, followed by the mean, standard deviation and log likelihood (rather than AIC): Github–Local

```
number of iterations= 10
summary of normalmixEM object:
          comp 1    comp 2
lambda  0.611002  0.388998
mu     23.011364 40.703294
sigma   1.720528  3.378663
loglik at estimate:  -28873.39
```

A plot of a sample drawn from a mixture of distributions does not always have visually distinct peaks; if f_1 and f_2 are normal densities with means μ_1 and μ_2, respectively, and both have the same variance σ^2, then the mixture density $f = 0.5f_1 + 0.5f_2$ will have a single peak if, and only if: $abs(\mu_2 - \mu_1) \leq 2\sigma$.

A study by Kaltenbrunner, Gómez, Moghnieh, Meza, Blat and López[952] analysed the pattern of user activity of the Slashdot technical community news site. The black curve in figure 9.18 shows the density of the number of accesses to one article in each minute after first publication (a total of 1,567 accesses).

A possible explanation for the multiple upticks in number of accesses, is the article being linked to from other websites, driving a fresh batch of readers to Slashdot. Which mixture of distributions might best fit the access times of this Slashdot article? The Poisson distribution is often used to model arrival times and is the obvious first choice, but in this particular case turns out not to provide the best fit.

Figure 9.18 shows several Normal distributions fitted to data, on a log scale, using functions from the rebmix and mixtools packages. The algorithms used by packages do not guarantee to find the globally optimal solution and differences in the mix of distributions selected can occur because of differences during the search process.

```
library("rebmix")

slash_mod=REBMIX(Dataset=list(data.frame(users=log(slash$users))),
                Preprocessing="histogram", cmax=5,
                Variables="continuous", pdf="normal", K=7:45)

plot_REBMIX_dist=function(dist_num)
{
y_vals=dnorm(x_vals, mean=as.numeric(slash_mod$Theta[[1]][2, dist_num]),
                 sd=as.numeric(slash_mod$Theta[[1]][3, dist_num]))
lines(x_vals, slash_mod$w[[1]][1, dist_num]*y_vals, col=pal_col[dist_num])
}


plot(work_den, main="", xlim=c(0, 10), ylim=c(0, 0.36),
        xlab="", ylab="Access density\n")
plot_REBMIX_dist(1); plot_REBMIX_dist(2)
plot_REBMIX_dist(3); plot_REBMIX_dist(4)
```

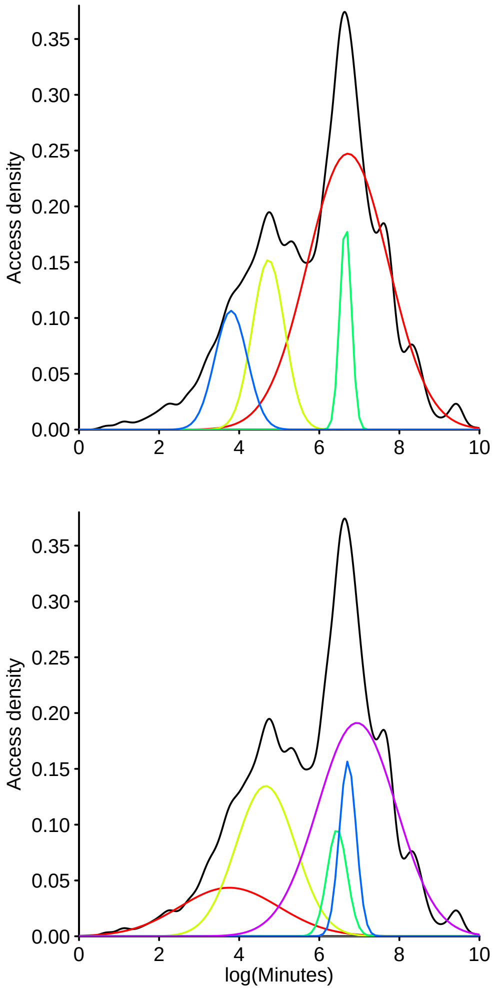


Figure 9.18: Density plots of accesses to one article on Slashdot, in minutes since its publication. The distinct Normal distributions (colored and fitted to the log of the data) contained in the mixture models fitted by the REBMIX (upper) and normalmixEM (lower) functions. Data kindly supplied by Kaltenbrunner.[952] Github–Local

Fitting a Normal distribution to log scaled data means that the sample actually has a Lognormal distribution. Is the Lognormal distribution a good representation for the processes driving readers to access Slashdot articles? As always in model building the answer depends on what the model is to be used for. If the purpose is to make predictions, the accuracy of prediction is of more interest than any underlying assumptions; if the purpose is to understand what is going on, then a theory containing processes generating Lognormal distributed behavior is needed.

It can take a lot of analysis, over many years, to settle on the distribution, or combinations of distributions, that best describes the measured properties of a system. The study of file-system characteristics[17] is an example of how researchers' ideas and models changed over time,[500, 885, 1283] becoming more sophisticated as more data became available, from various platforms,[440] and more analysis was performed.

9.3.3 Heavy/Fat tails

Heavy tailed is the term used to describe distributions where the majority of values occur a long way from the mean value (*fat tails* and *long tail* are also used).[vi] When the 80/20

[vi] The term *sub-exponential* is sometimes used to describe tails that decay slower than exponential and *super-exponential* for tails that decay faster than exponential.

rule applies the distribution is heavy tailed, and the frequency with which this rule is encountered suggests that such data is not rare. The `poweRlaw` package supports operations involving a variety of heavy tailed distributions, including power laws.

Averaging the performance of multiple subjects can produce values that are well fitted by a power law, while individual subject performance is well fitted by any of a variety of other distributions.[1316]

The Pareto distribution is the mathematical name of a particular instance of a heavy tailed distribution (sometimes going by the name *power law* in popular culture); Zipf's law is a particular instance of this distribution.

The mean value of a heavy tailed distribution may not exist (because it is infinite). Any finite dataset has a finite mean, and if a sample is drawn from a heavy tailed distribution, its mean value will jump around erratically.

It is more difficult to narrow down a distribution which best fits a sample drawn from a heavy tailed distribution (because several fit equally well), compared to one without a heavy tail; a sample may contain many values, but their density may be low because values are spread out over a long tail (rather than in a high density cluster around a central location).

Figure 9.19 is from a survey[885] of file sizes and shows that a small percentage of files account for most of the disk spaced occupied (the vertical line meets the bytes line where 89.9% of disk space has yet to be consumed, and the files line where 12.5% of files still remain to be accounted for). Another way of describing the situation is to say that there is a mass/count disparity, i.e., a few files occupy most of the space.

Care needs to be taken to separate out concepts that are popularly associated with power laws, e.g., *scale invariant*, which are a property of the distribution, not the generating process. The process generating data fitted by a power law can be remarkably random, e.g., the length of words in text produced by monkeys typing.[388]

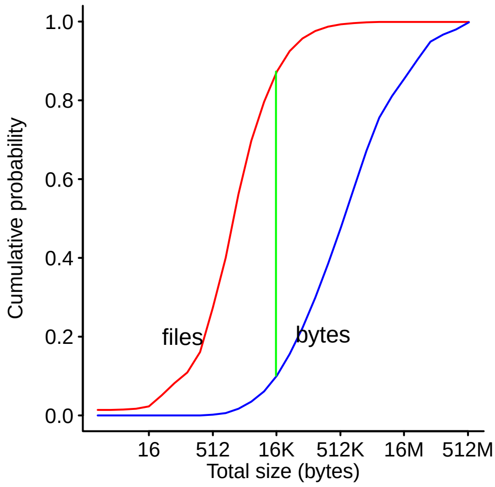


Figure 9.19: Cumulative probability distribution of files size (red) and of number of bytes occupied in a file system (blue). Data from Irlam.[885] Github–Local

9.4 Markov chains

A *finite state machine* (FSM) is a machine represented by a set of distinct states, connected by edges denoting the possible transitions that can occur when a given event occurs, such as when a particular character is input (FSMs are deterministic).

A *Markov chain* (MC) is also a machine represented by a set of distinct states connected by edges, but the possible transition is chosen at random based on the transition probability of each edge (the transition probabilities, out of any state, that is not an absorbing state, add to one); the next state only depends on the current state, i.e., the system is memoryless.

A Markov chain is a *discrete-time Markov chain* (DTMC), if the transition between states occurs at fixed time intervals; if the time interval between state transitions is not fixed, the Markov chain is a *continuous-time Markov chain* (CTMC) (the memoryless requirement means that transition times must have an exponential distribution). If the transition time depends on how long the system has been in the current state, it is a *semi-Markov process* (SMP).

Finite state machines provide a useful abstraction for modeling user interfaces. A study by Oladimeji[1393] investigated the user interface of the Alaris volumetric infusion pump (a medical device used for controlled automatic delivery of fluid medication, or blood transfusion, to patients); the user interface includes 14 buttons and an LCD display. Figure 9.20 shows the available transitions between states.

A FSM can be represented as a control flow graph. By using this representation functions in the `igraph` package can be used to answer questions such as: the maximum number of button presses needed to get to any state (12; the `path.length.hist` function returns a count of all possible path lengths), and the average number of presses to transition between any two states (4; using the `average.path.length` function).

If the behavior of a system (that can be represented using a FSM) is monitored, the probability of occurrence of every transition between states can be calculated. If the behavior represents typical user interaction with the system, then the probabilities can be used to create a Markov chain for this typical behavior.

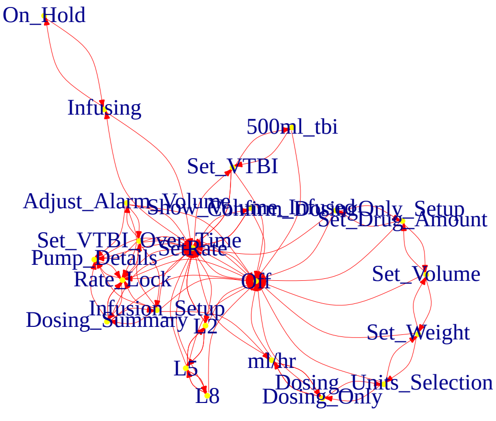


Figure 9.20: Graph of available state transitions for Alaris volumetric infusion pump (the button presses that cause transitions between states are not shown). Data kindly supplied by Oladimeji.[1393] Github–Local

A study by Tarasov, Mudrankit, Buik, Shilane, Kuenning and Zadok[1794] used data on the lifetime of source files in various systems, such as the Linux kernel, to generate realistic filesystem contents (for deduplication analysis). Figure 9.21 shows a Markov chain representing the life of source files in the Linux kernel (from being Initialised to new, through modified/unmodified to deleted and reaching the Terminal state). The measurement snapshot occurred at each of the 40 releases between versions 2.6.0 and 2.6.39, with an average of 23k files per snapshot; the time between releases is roughly constant, so this might be considered a discrete-time Markov chain.

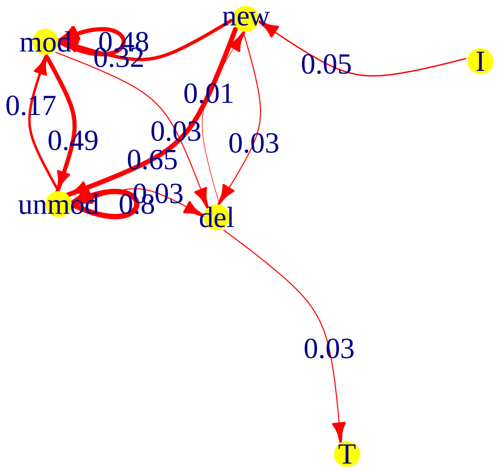


Figure 9.21: Discrete-time Markov chain for created/modified/deleted status of Linux kernel files at each major release from versions 2.6.0 to 2.6.39. Data from Tarasov.[1794] Github–Local

The `graph.data.frame` function assumes there is a link between the row values in two columns (`from` and `to` vertices) and builds a graph based on this assumption. The `V` and `E` functions access the vertices and edges of the graph and various attributes can be set and may be subsequently used by `plot`.

```
library("igraph")

atc=read.csv(paste0(ESEUR_dir, "probability/atc12-gra.csv.xz"), as.is=TRUE)
atc_gra=graph.data.frame(atc, directed=TRUE)

V(atc_gra)$frame.color=NA
V(atc_gra)$size=12 ; V(atc_gra)$color="yellow"
E(atc_gra)$arrow.size=0.5 ; E(atc_gra)$color="red"
E(atc_gra)$weight=E(atc_gra)$linux
E(atc_gra)$label=E(atc_gra)$weight/100

# layout.lgl outperforms the default layout for this graph
plot(atc_gra, edge.width=0.3*sqrt(E(atc_gra)$weight),
                        edge.curved=TRUE, layout=layout.lgl)
```

The algorithm used by `plot` to layout a graph makes use of randomization, which means that the layout returned by every call is different.

9.4.1 A Markov chain example

A study by Perugupalli[713, 1454] investigated the reliability of gcc, based on the reliability of its major subsystems. Information on the probability of a subsystem experiencing a failure was calculated, using the regression suite for gcc version 3.3.3 (which contains tests for 110 faults present in gcc version 3.2.3, out of 2,126 tests, of which 55 were traced back to the source code of a single subsystem; the others faults involved multiple subsystems). The researchers did not attempt to analyse failures involving more than one subsystem, and assumed that subsystems fail independently of each other.

Subsystems were identified by instrumenting gcc to count the number of calls between pairs of functions, made while executing the regression suite (this is not actually Markov chain-like behavior because the called functions return, which is not transition-like behavior). The 1,759 traced functions were manually assigned to one of 13 internal subsystems (e.g., parsing, tree optimization and register allocation),

The reliability of gcc version 3.2.3 might be estimated using:

$$R = 1 - \frac{F_c}{T_c} = 1 - \frac{110}{2126} = 0.948$$

where: F_c is the number of source files that it did not correctly compile and T_c is the total number of files compiled.[vii]

This approach has the advantage of being simple to calculate, but does not provide any information on the impact of individual subsystems on overall reliability, for instance, what is the sensitivity of overall system reliability to behavioral changes to one subsystem?

The probability of reaching subsystem n from subsystem 1 after k transitions is Q^k (where Q is the matrix of transition probabilities). Summing over all transitions (using an infinite upper bound for the total number of transitions simplifies the mathematics), we get:[1825]

$$S = \sum_{k=0}^{\infty} Q^k = (I - Q)^{-1}$$

[vii] The calculated reliability is very low because it is based on compiling a test suite of short code samples designed to reveal faults.

where: I is the identity matrix. The expression $(I-Q)^{-1}$ is easily calculated (i.e., inverting the result of a matrix subtraction). The matrix S, is known as the *fundamental matrix*, and can be used to calculate a variety of properties of systems modeled by the Markov chain.

The composite and hierarchical methods are two techniques for combining information on subsystem usage (i.e., subsystem transition probabilities and subsystem reliability, calculated using the above formula), to calculate the reliability of a complete system:

- composite method:[342] this calculates the probability of a successful transition between each subsystem, by multiplying the transition probabilities of each subsystem by the probability of the subsystem executing successfully. These individual successful transition probabilities are used to calculate the successful transition probability from the initial subsystem to the final subsystem (i.e., the system's fundamental matrix). The estimated reliability calculated for gcc is 0.9972,[viii] (see Github–reliability/gcc-reliability.R).
- hierarchical method: if R_i is the reliability of a subsystem, the probability of all executions of that subsystem being successful is $R_i^{N_i}$, where N_i is the number of transitions to subsystem i during one execution of the system. Assuming that subsystems fail independently, the expected value of system reliability is:

  $$R = E\left[\prod_{i=1}^{n} R_i^{N_i}\right]$$

  Assuming subsystems are highly reliable, and the variance in the number of subsystem transitions is very small, the first order Taylor approximation can be used:

  $$R \simeq \prod_{i=1}^{n} R_i^{V_i}$$

  where: $V_i = E[N_i]$ is the expected number of times a transition occurs to subsystem i, during a single execution of the complete system; V_i is obtained by solving:

  $$V_i = q_i + \sum_{j=1}^{n} V_j p_{ji}$$

  where: q_i is the probability that execution starts with subsystem i, and the p_{ji} are obtained from the subsystem transition probability matrix (see Github–reliability/gcc-reliability.R).

The `markovchain` package supports discrete time Markov chains, and the `msm` package supports continuous time through the use of multi-state models.

9.5 Social network analysis

The popularity of web based social networks has made the mathematics of social network analysis a fashionable research topic. Unfortunately many published papers involve little more than claiming to have found a power law, with only pretty pictures and hand waving to show. Table 7.1 is an example of the kind of descriptive statistics encountered in social network analysis.

Social networks are represented as graphs, and the `igraph` package supports reading many graph data representation formats, along with a wide range of operations and analysis on graphs.

A study by Canfora, Cerulo, Cimitile and Di Penta[293] analysed the developer's mailing lists for FreeBSD and OpenBSD, to obtain information on what they called *Cross-System-Bug-Fixings*; the data contains information on 861 unique developers sending email and 1,062 unique developers receiving email. Both FreeBSD and OpenBSD were forked from a common base and not only continue to share common code but faults fixed in one are often applied, some time later, to the other. Figure 9.22 was produced using code very similar to that used for the Markov chains in figure 9.21.

Many real world collections of linked node contain subgroups (e.g., clusters of developers or related code modules), and there are a variety of algorithms for detecting these subgroups. Care needs to be exercised in interpreting the clusters returned by these algorithms, as there may be many distinct high-scoring solutions, and a clear global maximum may not exist.[697]

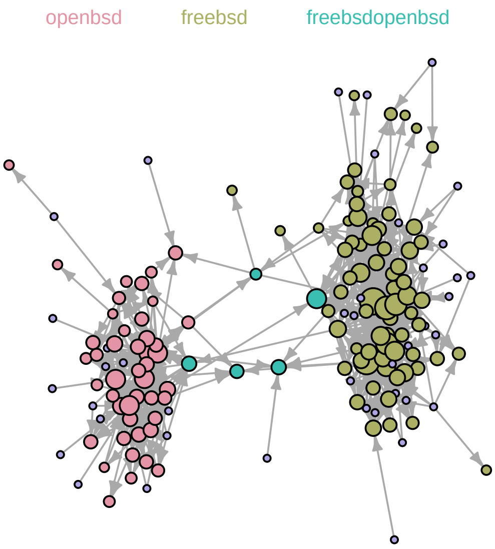


Figure 9.22: Directed graph of emails between FreeBSD and OpenBSD developers, plus a few people involved in both discussions, with developers who sent/received less than four emails removed. Data from Canfora et al.[293] Github–Local

[viii] If the 55 fault count used in this analysis is plugged into the simple formula used above, the reliability estimate is 0.974.

9.6 Combinatorics

The analysis of some systems makes it necessary to consider combinations of various items, and there is a need to enumerate all possible sequences, to find the total number of different sequences of items that could occur (or other related questions). The mathematics used to solve this kind of problem is known as *combinatorics*.

A few of the functions frequently used in combinatorial problem solving are included in R's base system, including:

- the choose function takes two arguments, n and k and returns the value $\frac{n!}{k!(n-k)!}$, often written as $\binom{n}{k}$; the number of ways of selecting k items from n items,
- the combn function takes two arguments, x and k and returns an array containing all combinations of the elements of x taken k at a time.

When an item is drawn, with replacement, from a pool of items the probability of drawing the same item again is unchanged, when drawing without replacement the probability will decrease by the appropriate amount. An item is distinct if it is treated as being different from all other items in the pool (even when drawing with replacement), e.g., there are four items in the pool `x=c("a", "a", "b", "c")`, but only three of them are distinct.

Table 9.1 show how the `iterpc` function in the `iterpc` package can be used to generate sequences based on the distinctness of items and whether they are drawn with replacement or not.

		Distinct	
		True	**False**

Replacement	**True**	`I=iterpc(5, 2,`
		`replace=TRUE)`
			`x=c("a", "a", "b", "c")`
			`I=iterpc(table(x), 2,`
			`replace=TRUE)`

	False	`I=iterpc(5, 2)`	`x=c("a", "a", "b", "c")`
			`I=iterpc(table(x), 2)`

Table 9.1: Example `iterpc` calls generating particular kinds of sequences of length two (by passing the value returned to `getall`, e.g., `getall(I)`).

The treatment of item ordering is another factor, when considering all possible permutations; is the ordering of items significant or not, e.g., are the sequences `a,b` and `b,a` treated as different or equivalent? When the ordering of items is significant calls to `iterpc` need to set the optional argument `ordered` to `TRUE`, e.g., `I=iterpc(5, 2, ordered=TRUE)`.

9.6.1 A combinatorial example

This example illustrates the kind of detailed analysis needed to solve a practical combinatorial problem.

A study by Jones[923] investigated developer preferences for ordering members within C `struct` types. The hypothesis was that members having the same type are likely to be grouped together within the same `struct` type.

The data contains enough instances of `struct` types containing between three and eight members, for the sample to be analysed with a reasonable level of confidence.[ix]

If a `struct` contains n members, the number of possible member sequences is $n!$. However, we are only interested in member types and don't care about permutations of members having the same type. The number of different member type sequences is $\frac{n!}{n_1!n_2!\cdots}$ where $n = n_1 + n_2 + \cdots$ and n_1, n_2, etc are the number of members having a given unique type.

Taking the example of a `struct` containing four members, two of type `x` and two of type `y` the possible sequences of member types within a `struct` type are:

[ix] The number of `struct` types containing a given number of members decreases approximately logarithmicly with increasing number of members,[919] i.e., most member sequences are relatively short.

```
xxyy xyxy yxxy xyyx yxyx yyxx
```

and if two members are of type `x`, one of type `y` and one of type `z`, the possible member type sequences are:

```
xxyz xxzy xyxz xyzx xzxy xzyx yxxz yxzx yzxx zxyx zxxy zyxx
```

In the first case members are grouped together in $\frac{1}{3}$ of cases and in the second, in $\frac{1}{2}$ of cases.

If there are t different types, there are $t!$ possible unique sequences of types. If the ordering of `struct` members is random, the probability of encountering a definition in which all members having the same type are grouped together is: $\frac{t!}{\frac{n!}{n_1!n_2!\cdots n_t!}}$. For the two examples above the probabilities of encountering a member ordering, where identical types are grouped together, are: $\frac{2!}{\frac{4!}{2!2!}}$ and $\frac{3!}{\frac{4!}{2!1!1!}}$ (which is already known from enumerating out all possible sequences).

When a `struct` contains four members, as in the above examples, it is not possible to distinguish between a developer intentionally choosing an order and random selection. For `structs` types containing five members, the probability of random selection of member order, grouping together the same member types is high; see the fifth column in table 9.2.

Total members	Type sequence	structs seen	Grouped occurrences	Random probability	Occurrence probability
4	1 1 2	239	185	0.50	2.83×10^{-18}
4	1 3	185	146	0.50	4.75×10^{-16}
4	2 2	98	61	0.33	4.58×10^{-09}
5	1 1 1 2	57	50	0.40	1.03×10^{-13}
5	1 1 3	94	61	0.30	3.13×10^{-12}
5	1 2 2	86	49	0.20	5.18×10^{-14}

Table 9.2: Various forms of `struct` types containing a given number of members, one possible type grouping, number of actual `struct` types measured, number having grouping, probability that one type will contain this grouping and probability that the number grouped, out of total seen, will be so grouped. Data from Jones.[923] Github–Local

Table 9.2 shows source code measurement counts and calculated probabilities for `struct` types containing four and five members: the column *Total members* lists the number of members in the type, *Type sequence* is a possible grouping of member types for a given number of member types, *structs seen* is the number of measured `structs` containing the given number of members/types, *Grouped occurrences* is the number of measured `structs` having the grouping listed in the first column, *Random probability* is the probability of this grouping occurring randomly in one `struct` declaration containing the given number of members and types, *Occurrence probability* is the probability of *Grouped occurrences* out of *structs seen* occurring, when the probability of a single instance occurring is *Random probability*.

This analysis shows it is not possible to confidently distinguish between random and intentional ordering, for individual `struct` types. However, programs contain many such type definitions, and if we label each one "Yes" or "No", depending on whether their member types are grouped or not, this list of Yes/No labels has a binomial distribution, and the probability of a given number of Yes/No labels occurring through chance can be calculated.

Taking the example of a `struct` containing four members, two of type `x` and two of type `y`, the probability of a single random occurrence of this sequence is $\frac{1}{3}$; the sample analyzed contains 98 `struct` types with four members having two distinct types, of which 61 have this sequence of member types (see columns 3 and 4 in table 9.2). The probability of this occurring is calculated using `pbinom(61-1, 98, 1/3, lower.tail=FALSE)`, whose value is `4.58272e-09` (the `lower.tail=FALSE` option is used because we are interested in the probability of seeing 60 or more occurrences).

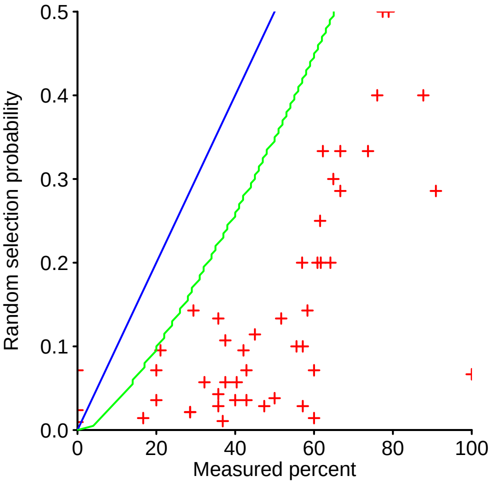


Figure 9.23: Expected probability of a single instance (y-axis) against the probability of a measured **struct** type having grouped member types (x-axis); when both probabilities are the same points will be along the blue line. Data from Jones.[923] Github–Local

Figure 9.23 shows the measured percentage of `struct` types whose members are grouped by type (red pluses), and the percentage that would occur with random ordering (blue line). The green line is the 99.9% probability bound, for the likelihood that 100 `structs`, all sharing the same member types, will all have their members grouped by type when member ordering is chosen at random. The distance of the red crosses from the 99.9% bound shows that grouping of members by type is very unlikely to have been driven by random selection.

9.6.2 Generating functions

Generating functions are discussed here purely to inform readers about a powerful technique, which is significantly different from the traditional approach to solving probability problems using factorials; this technique is capable of solving problems that appear to be otherwise intractable. If it is not possible to derive an expression specifying how many possibilities can occur in some situation, then a search for the appropriate generating function may provide an answer.

Generating functions are starting to be covered in texts on probability; some mathematical sophistication is required.

A generating function is a polynomial $a_0x^0 + a_1x^1 + \cdots + a_nx^n$, where the coefficients a_n encode information about the quantity of interest.

The following is a simple example that could just as easily be calculated using factorials, but illustrates the idea. How many ways can five items be selected, if `A` can be selected 0 or 1 times, `B` can be selected 0, 1 or 2 times and `C` can be selected 0, 1, 2, 3 or 4 times? The generating function is (see the suggested reading for why this works):

$$(1+x)(1+x+x^2)(1+x+x^2+x^3+x^4) = x^7+3x^6+5x^5+6x^4+6x^3+5x^2+3x+1$$

the coefficient of x^5 is 5, so five different items orderings are possible.

A more complicated example is when items have a particular value and sequences that sum to a specific total are required. If `A` is worth 1, `B` is worth 3 and `C` is worth 5, the generating function is:

$$(1+x+x^2+\cdots)(1+x^3+x^6+\cdots)(1+x^5+x^{10}+\cdots) = 7x^{11}+7x^{10}+6x^9+5x^8+4x^7+4x^6+\\ 3x^5+2x^4+2x^3+x^2+x+1$$

the coefficient of x^{10} is 7, so there are seven different ways of selecting items that sum to ten.

The `polynom` package supports the symbolic manipulation of polynomials.

Chapter 10

Statistics

10.1 Introduction

Is a pattern of interest present in a population?

Statistics provides information about a population, based on a measurement sample drawn from that population.

The developer input to statistical analysis process is their domain knowledge, which may suggest patterns of behavior to search for, and provide one or more interpretations to any patterns that are found (the feedback given may be that the pattern found is not interesting).

The output from statistical analysis should be treated as a guide, not as a definitive statement.

Correlation does not imply causation, a common mantra that is always worth repeating.

Traditionally, statistical techniques have had to be practical to perform manually. This has resulted in general statistical problems being split into a profusion of specific subproblems, and the creation of techniques tailored to handle each case. Doing statistic analysis this way, involved mapping the sample characteristics to a particular subproblem and then applying the corresponding technique. Computer availability makes it practical to apply general solution techniques and general, powerful and robust statistical techniques are available;[542] however, many existing users of statistical techniques have simply switched from manual to computer based calculation of familiar historical techniques, without appreciating the original design rational for these techniques. Many statistical techniques appearing in this book are impractical to apply manually (e.g., the bootstrap) a computer is required.

The results from data analysis may vary with the person doing the analysis;[1684] for instance, people may use a technique because it is the one they know how to use, rather than the technique best suited to the data being analyzed.

Existing books often invest effort massaging data into a form that permits the use of techniques that depend on the data having a Normal distribution (also known as a _Gaussian distribution_[i]). The reasons for this are historical (assuming Normality made the analysis tractable in the days before computers), and data in the Social sciences (early adopters of statistical techniques and a major market for statistical books) appearing to be drawn from a Normal distribution (despite the claims made, data in this field often does not have a Normal distribution[1257]). It might be said that nobody ever got fired for assuming a Normal distribution.

Measurements of software engineering processes often produce values that are not drawn from a Normal distribution; the Exponential and Poisson distributions are relatively common; measurement samples that are best described by a Normal distribution do occur, but they do not have the dominant market share encountered in other, non-software related, domains (e.g., the social sciences).

The input to statistical analysis is a sample and usually some expectations of behavior; the expectations may be explicit (e.g., measurements are independent of each other) or

[i] This book uses the term Normal because it appears to be more widely used.

implicit (e.g., the choice of a statistical technique that only produces reliable results for samples drawn from a population having certain characteristics).

The possibility for detecting patterns that might be present in a sample depends on the quality and quantity of measurement data:

- quality: noise in the measurement process and errors in post measurement processing (e.g., incorrect conversion of file formats or inaccurate calculations of values derives from the raw data) are some of the problems that affect data quality,
- quantity: the number of measurements impacts the power and significance of statistical tests, and the confidence bounds on the results of statistical analysis.

Finding a pattern in the data having the desired level of statistical certainty, moves the discussion on to the practical engineering consequences of what has been found, e.g., mountain or molehill. A discussion of practical engineering consequences of patterns is outside the scope of this book.

10.1.1 Statistical inference

The most commonly used statistical inference technique makes use of *frequentist* methods, i.e., how often events occur and long-run averages. All techniques have problems associates with their use and frequentist, being the most widely used, has the greatest number of detractors; a common problem is misuse of the concept of p-value; any widely used technique will have a common failure mode, simply because of varying skills of the people using it. The p-value is the fall-guy of the frequentist approach to statistical analysis.

The frequentist approach is the technique predominantly used in this book because it is commonly used in statistical books and articles; it is used by most R packages and readers are likely to encounter it when interacting with other people involved in analysing and using data.

Another technique is *Bayesian statistics*, which is growing in popularity; some R packages use this approach internally. A Bayesian approach has the potential to extract more information from data, by making use of information about prior beliefs. What is known as the *prior*, is a reasonable value for the probability of the event occurring, estimated prior to any measurements being made (the measurements get factored in later); the selection of a suitable prior opens the door to the bias of opinion and policy guidelines,[230] e.g., a Bayesian approach to deciding whether the accused is guilty runs into the problem that many legal systems assume people are innocent until proven guilty (i.e., the prior is zero), a belief that percolates through calculations to always produce a not-guilty result.

A study by Furia[629] reanalyzes several software engineering datasets using Bayesian techniques.

Maximum likelihood estimation, MLE, is a technique for finding the set of parameters for a model that make the observed data most likely to have occurred.

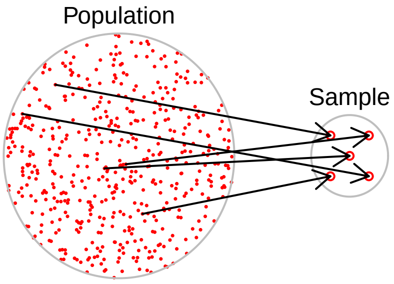


Figure 10.1: Example of a sample drawn from a population. Github–Local

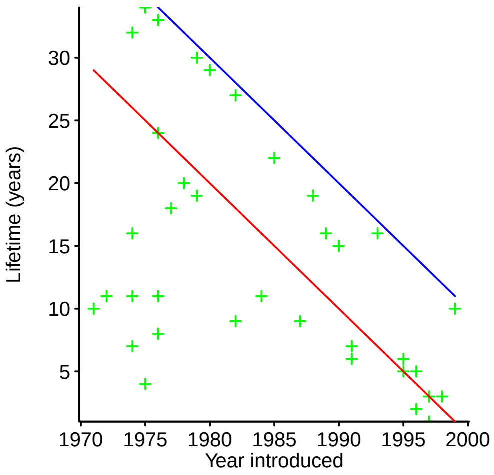


Figure 10.2: Date of introduction of a cpu against its commercial lifetime; processors ceasing production in 2000 or 2010 would appear along one of the lines. Data from Culver.[412] Github–Local

10.2 Samples and populations

It may not be practical to measure every member of a population, and the subset of the population measured is known as a *sample*; see figure 10.1.[ii] Depending on the question being asked, a set of measurements may be a population or a sample. For instance, measurements of one particular program yields the parameters of a population when the questions being asked concern just that one program, but they become the statistics of a sample when generalizing the findings to questions about other programs (including future versions of the one measured).

A sample is selected as a proxy for the entire population (experimental subjects are discussed in section 13.2.1). There are a variety of sampling techniques, including:

[ii] The term *statistic* applies to values calculated from a sample, while the term *parameter* applies to values calculated from a population. In some equations the value $N-1$ is used, when N might appear to be more appropriate. A mathematical distinction occurs between samples and populations, in that sample estimates are based on degrees of freedom of the sample, i.e., the number of members in the sample minus one, while population parameters are based on the number of members in the population, i.e., N.

- a *survey sample* is collected when the items to be measured (often via a questionnaire) are selected from a population assumed to share (unknown) fixed characteristics. The measured characteristics of the random sample, drawn from this fixed population, are used to estimate the characteristics of the population. The analysis of a sample obtained via a survey is *design-based* (rather than *model-based*). See section 13.4 for a discussion of questionnaire based surveys,
- a *prospective* study collects data as events unfold. Figure 10.2 shows the date of introduction of a cpu against its commercial lifetime, in years.[412] Processors that ceased production in 2000 or 2010 would appear along one of the two colored lines,[iii]
- a *retrospective* study collects data after events have taken place,
- a *convenience sample*, as its name implies, makes do with what is available,
- *snowball sampling*, or *chain sampling* starts with an initial list of subjects, who are asked to propose other subjects whom to them, with the process iterating until the number of new subjects falls below some threshold,
- stratified sampling divides the population into what are known as *strata*, with the strata chosen such that similar cases tend to cluster within each one; each of these strata are then sampled (using, say, random sampling) to produce the final sample (which is a set of distinct stratum, see figure 10.3),
- sequential sampling is covered in section 13.2.4,
- interval sampling divides the measurement interval into a series of fixed points and samples at just these points. The width of the sampling intervals puts a lower bound on the behavior that can be resolved. An experimental study[iv] by Kistowski, Block, Beckett, Lange, Arnold and Kounev[1000] measured power consumption, using programs from SPEC's Server Efficiency Rating Tool, at load level increments of 2% (crosses) and 10% (lines); see figure 10.4. A cost/benefit analysis would compare the greater accuracy obtained using finer measurement intervals against the likelihood of sudden jumps in the response value, that could have a noticeable impact on the results.

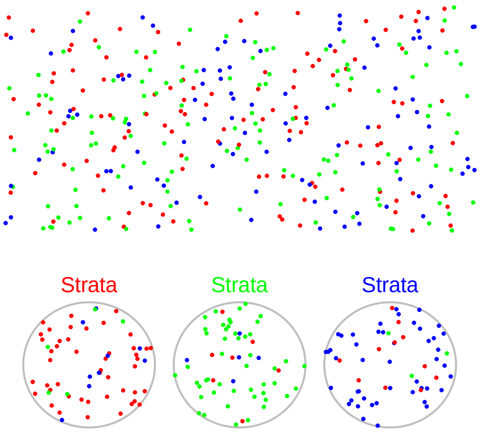


Figure 10.3: A population of items having one of three colors, along with samples of the three strata (imperfect item selection introduces noise in the samples). Github–Local

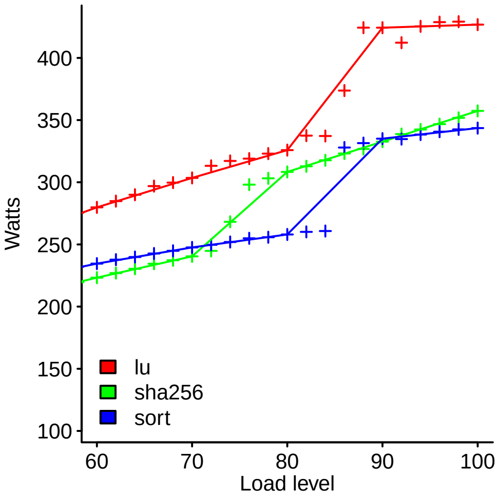


Figure 10.4: Power consumed by three SERT benchmark programs at various levels of system load; crosses at 2% load intervals, lines based on 10% load intervals. Data kindly provided by Kistowski.[1000] Github–Local

Occasionally the subjects of interest are not present in the sample. For instance, the damage experienced by aircraft returning from combat, during the second world war, was analysed with a view to improving aircraft survival rate. A statistician involved in the analysis pointed out that important subjects were missing from the sample,[1185] aircraft that had not returned. The return of a damaged aircraft provides evidence that the damaged areas are not critical to survival; it was those areas not damaged in returning aircraft that are likely to be critical to survival.

Guy[753] proposed a *strong law of small numbers*, "There aren't enough small numbers to meet the many demands made of them.", listing 35 examples of numeric patterns found in samples calculated using small integer values that disappear when larger integer values are used. The greater number of large values reduces the likelihood of coincidental correctness.

Figure 10.5 shows the four connected statistical characteristics of a sample; given the values of three of them, the fourth can be calculated.

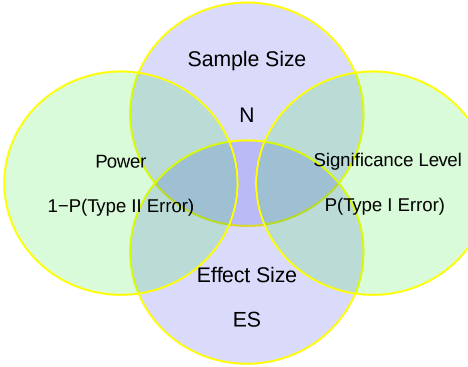


Figure 10.5: The four related quantities in the design of experiments; given three, the fourth can be calculated. Github–Local

While gathering a representative sample of the population as a whole is a common requirement, sometimes samples having other characteristics are of interest, e.g., being diverse,[1326] or intended to maximise the number of faults found.[1238]

The algorithm used to select the members of a sample can be non-trivial, even for uniform sampling, e.g., uniform distribution of points within a circle, or uniform sampling from Kconfig feature models.[1388]

10.2.1 Effect-size

Effect-size is the degree to which the characteristic of interest is present in the population (from which a sample is drawn), e.g., if we are interested in the difference in the performance of developers before and after attending a training course, how big is the difference (answering this question may be the reason for obtaining measurements)?

The question to ask about a calculated effect-size is: "Does it matter?" The larger the effect-size the more likely it is to be of interest in practice; in some cases a small effect-size may be of interest (e.g., a small difference multiplied over a large population can

[iii] Email discussion with the author confirmed that the data had not been updated since 2010.

[iv] The study was experimental because it did not meet all the requirements for an official SERT run.

have a large impact), while in other cases only a large effect-size is of interest (e.g., when the population is small a large effect-size may be needed to have a large impact).[v]

Smaller effect-sizes are likely to be more costly to detect because more measurements are needed to isolate small effect-sizes compared to larger ones.

Figure 10.6 shows how percentage differences in the presence of a condition in a population can have a dramatic effect on the false positive rate (in red), for the same statistical power and p-value.

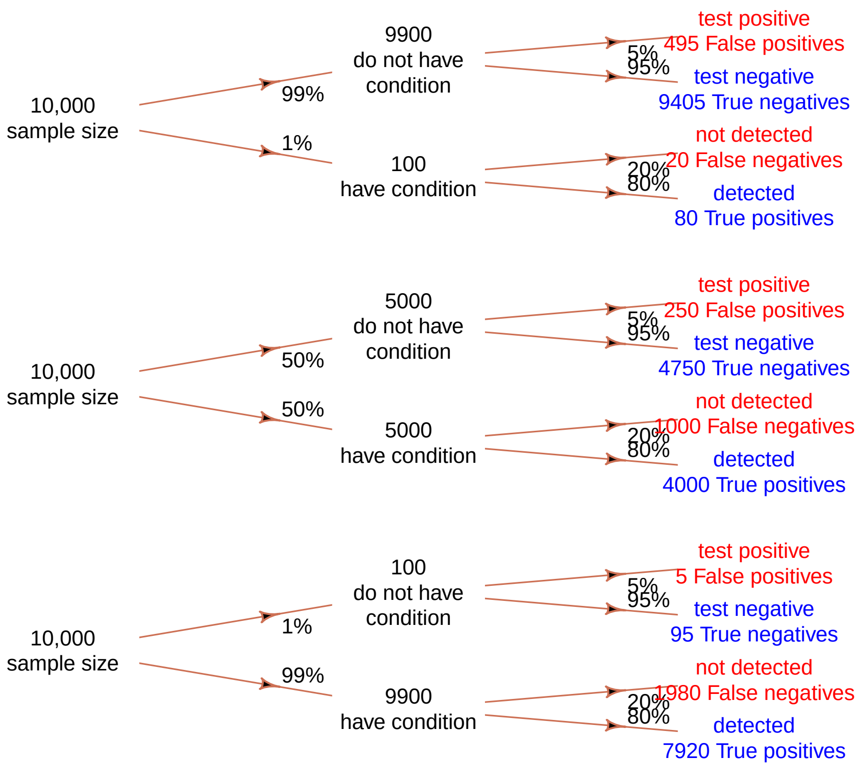


Figure 10.6: Examples of the impact of population prevalence, statistical power and p-value on number of false positives and false negatives. Github–Local

Methods for calculating effect-size depend on the kind of analysis being performed on the sample,[535] and include the following:

- correlation, e.g., the Pearson correlation coefficient, is a measure of effect-size,
- combining information on the mean and standard deviation of two samples into a single value. For instance, Cohen's d is one measure used when samples have similar standard deviations, and is given by: $d = \frac{\mu_1 - \mu_2}{\sigma_{pooled}}$. There are a variety of effect-size calculations associated with Cohen's name.

  Figure 10.7 illustrates how differences in mean and standard deviation, of two distributions, result in a given Cohen's d,

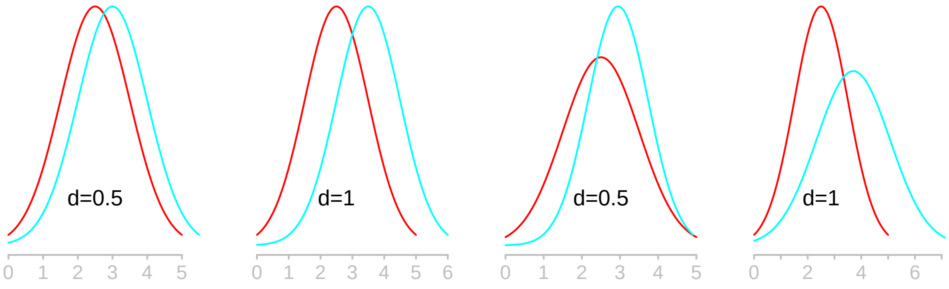


Figure 10.7: Visualization of Cohen's d for two normal distributions having different means and the same standard deviation (two left), and different mean and standard deviations (two right). Github–Local

- odds ratio (i.e., $odds = \frac{p}{1-p}$), that is, the ratio of the odds of an event occurring in one sample divided by the ratio of the same event occurring in the other sample (perhaps a control group).

[v] Statistical books[370] and papers sometimes concern themselves with questions of where to draw the lines that delimit large/medium/small effect-sizes, an approach that might be applicable when researchers are more interested in publishing papers than making useful discoveries.

10.2.2 Sampling error

If the reader agrees that sampling error is an important issue, this section can be skipped. Otherwise, read on and be frightened into agreeing.

The Central Limit theorem is a statement about the mean value of samples drawn from a population. If the population has a finite variance (power laws with an exponent between zero and two have an infinite variance), then the distribution of sample means converges to a Normal distribution as the sample size, N, increases (it does not matter what distribution the population has, it is the distribution of sample means that converges to the Normal).

How quickly does the distribution of sample means converge? The Berry-Esseen theorem gives the best known estimate of convergence of the distribution of the mean of independent, identically distributed, variables to a Normal distribution:

$$|F_n(x) - \Phi(x)| \le \frac{0.34(\rho + 0.43\sigma^3)}{\sigma^3\sqrt{N}}$$

where: F_n is the cumulative distribution function of the means, Φ the cumulative distribution function of a Normal distribution, ρ the third moment of x (and less than infinity), σ the standard deviation and N the sample size.

The only parameter available for influencing the error is the number of measurements; the error is proportional to: $\frac{1}{\sqrt{N}}$, e.g., to halve the error in the sample mean, the sample size needs to increase by a factor of four.

Figure 10.8 shows the distribution of mean values for samples drawn from three different distributions (using two sample sizes); the vertical lines are 95% confidence bounds.[vi]

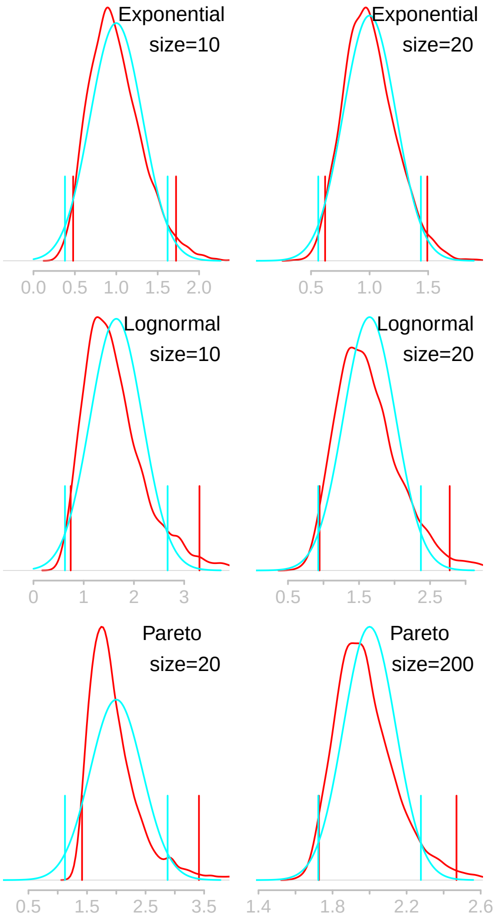


Figure 10.8: Distribution of 4,000 sample means, for two sample sizes, drawn from exponential (upper), lognormal (center) and Pareto (lower) distributions, vertical lines are 95% confidence bounds. The blue curve is the Normal distribution, predicted by theory. Github–Local

A study by Chen, Chen, Guo, Temam, Wu and Hu[336] measured the performance of programs in the SPEC CPU2006 benchmark using 1,000 sets of input data for each program. As an exercise in sampling let's assume we only have access to three of a possible 1,000 input datasets, what range of execution times might we expect to see from processing just three datasets?

Figure 10.9 was obtained by randomly sampling three items from the population of 1,000 and repeating the process 100 times. The red cross is the sample mean, and the vertical brown lines each sample's standard deviation; the blue line is the mean for the population of 1,000 input sets and the green lines the bounds of this population's standard deviation.

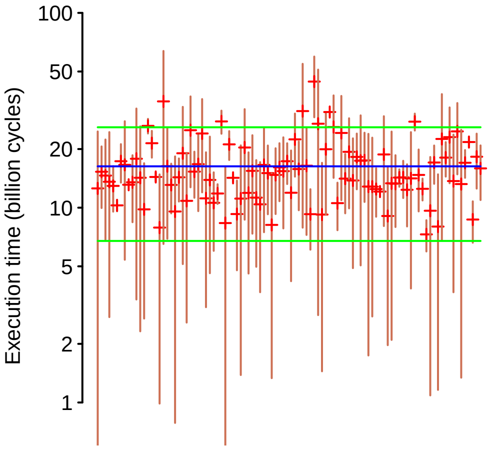


Figure 10.9: Mean (red) and standard deviation (brown line for each sample; not symmetrical because of log scaling) of samples of 3 items drawn from a population of 1,000 items (whose mean shown by blue line and standard deviation by green lines). Data kindly provided by Chen.[336] Github–Local

Figure 10.10 shows the distribution of sample means for sample sizes of 3 and 12 items. As expected, the larger samples show less variation in the mean value.

Sources of noise (i.e., random variability) in a sample include the following:

- measurement error caused by imperfect tools used to make measurements, which can include coding mistakes and the definition of what is being measured, e.g., lines of code,[1683]
- demographic variability, e.g., measurements of particular kinds of programs, or developers working in a single location or for one company,
- environmental variability is the sea in which developers swim, or have swum in the past, e.g., company culture or habits acquired from early teachers.

Figure 10.11 shows the number of commits to glibc[696] for each day of the week, separated out by year. The plot in the top left shows daily totals over all years. The combined plot suggests that most commits occur near the middle of the week, with the number falling off towards the beginning and end of the week. However, the yearly plots rarely show anything like this pattern; is any interpretation of the pattern of commits in the combined plot a just-so story?

10.2.3 Statistical power

If an effect exists, and an experiment is performed to measure it, what is the likelihood that the effect will be detected? The numeric answer to this question is known as the *statistical power*, of the experiment. The *power* of a statistical test is its ability to detect a difference when one actually exists in the data. Failing to detect an effect, when one exists, is known as making a *Type II error*, or more commonly as a false negative (β is

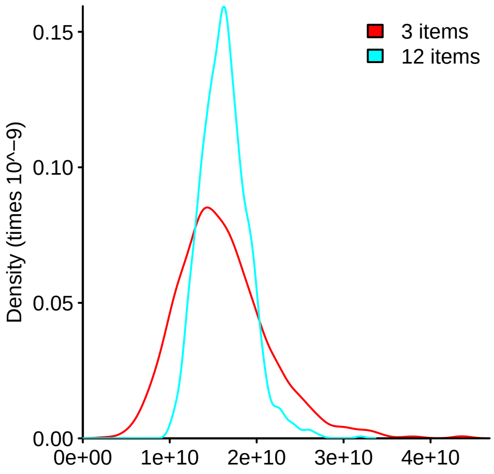


Figure 10.10: Density plot of mean of samples containing 3 or 12 items randomly selected from a data set of 1,000 items; process repeated 1,000 times for each sample size. Data kindly provided by Chen.[336] Github–Local

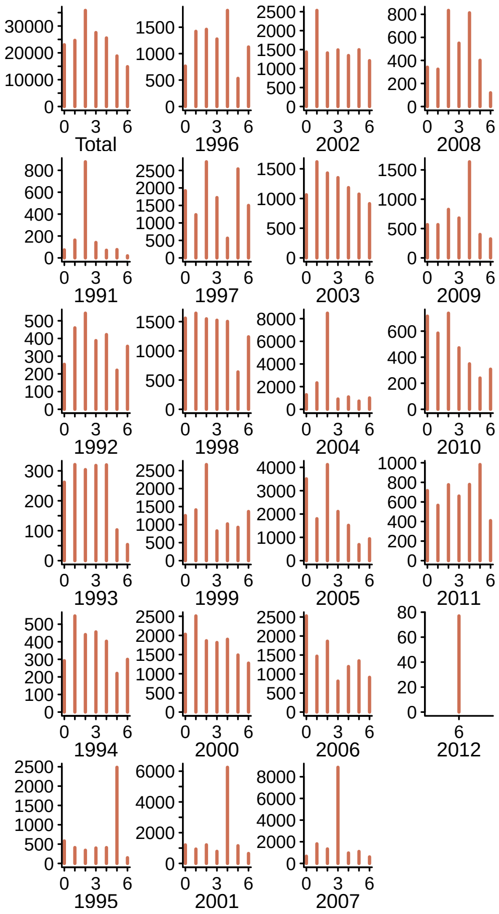


Figure 10.11: Number of commits to glibc for each day of the week, for the years from 1991 to 2012. Data from González-Barahona et al.[696] Github–Local

commonly used to denote the Type II error rate). Techniques for reducing Type II errors include:

- being willing to accept a larger *Type I error*, or more commonly as a true negative (α is commonly used to denote the Type I error rate),
- sampling from a population thought to have a higher probability of containing the sought after characteristics. For instance, Vasa[1864] excluded releases with less than 30 changed classes in a study of class change dynamics. If a subset of a population is selected to maximise detection rate, care must be taken to ensure that any statement of statistical power refers to the subset population, not the larger population from which it was subsetted,
- increasing the number of measurements made, i.e., sample size.

Figure 10.12 is an example showing the distribution of measurements in two populations: X (red) and Y (green) (e.g., the time taken to execute all possible programs, with all possible input, on two different computers). The upper and middle plot only differ in mean value, while the middle and lower plot only differ in standard deviation. The false positive rate, α, is shaded in red, and the false negative rate, β, in green. The two rates are connected in that increasing one decreases the other, and vice versa.

When there is a large overlap between samples (middle plot), most of the measurements in either sample have values that suggest they could have drawn from the other sample. In the upper plot, the difference in the sample means makes it more likely that measurements from Y will have values that appear to have been drawn from a different distribution, than samples from X.

The area of the unknown distribution excluding β (i.e., $1 - \beta$), is known as the power of the test.

A power of 80% is often quoted[370] as being an acceptable lower limit of a test having a high power, just like 5% is often quoted as an acceptable significance level in many contexts.

If there is a need to estimate whether an effect exists (e.g., one computer is faster than another, or a new algorithm uses less memory), before an experiment is run the question to ask is whether a difference (if it exists) is likely to be detected using the available resources (e.g., time and effort needed to obtain a measurement sample). A statistical power calculation shows the tradeoffs that can be made between sample size and probability of detecting an effect (assuming information on population mean, standard deviation and estimated differences between two or more samples).

The `pwr` package supports power analysis calculations for a variety of standard statistical tests. The functions are passed values for three sample characteristics and return the value of the fourth (see fig 10.5).

A study by Syed, Robinson and Williams[1784] investigated variations in the number of intermittent failures experienced, when using the Firefox browser, at different processor speeds, system memory and hard disc sizes. A total of 11 known coding mistakes, causing intermittent failure (four of these did not produce fault experiences) and nine different hardware configurations were selected. The conditions expected to cause each mistake to result in a fault being experienced were created, and Firefox was executed 10 times with each hardware configuration. Table 10.1 shows the number of each fault experienced with each hardware configuration.

This experiment failed to detect a connection between hardware configuration differences and faults experiences. What is the likelihood that if a connection existed, this experiment would have detected it. Alternatively, how large would the connection need to be for this experiment to detect it?

Analyzing the statistical power of an experiment involving a difference in proportions (i.e., failures before and after) requires an estimate of effect size (calculated from the proportion of failures before and after a change of hardware specification), the number of runs (10 in this case), and the desired p-value (e.g., 0.05). In this study, there were multiple changes of hardware specification and to keep things simple this analysis calculates the power for one change.

Does a hardware change cause more or fewer faults to be experienced? Without a theory providing a believable rationale for more/less, it has to be assumed that either could occur. In other words, a two-sided test is required.

[vi] There will be fluctuations in the values drawn to create each sample.

Mhz-Mb-Gb	124750	380417	410075	396863	494116	264562	332330
667- 128- 2.5	4	10	6	5	2	3	5
667- 256-10	4	8	8	6	4	3	8
667-1000- 2.5	4	7	3	4	3	1	8
1000- 128-10	3	10	3	6	0	1	1
1000- 256- 2.5	3	9	0	6	0	1	2
1000-1000-10	2	9	4	5	0	0	1
2000- 128- 2.5	0	10	5	6	0	0	0
2000- 256-10	2	8	5	7	0	0	0
2000-1000-10	1	7	3	5	0	0	0

Table 10.1: Number of times, out of 10 execution, a known (numbered) coding mistake resulted in a detectable failure of Firefox running on a given hardware configuration (cpu speed-memory-disk size). Data from Syed et al.[1784]

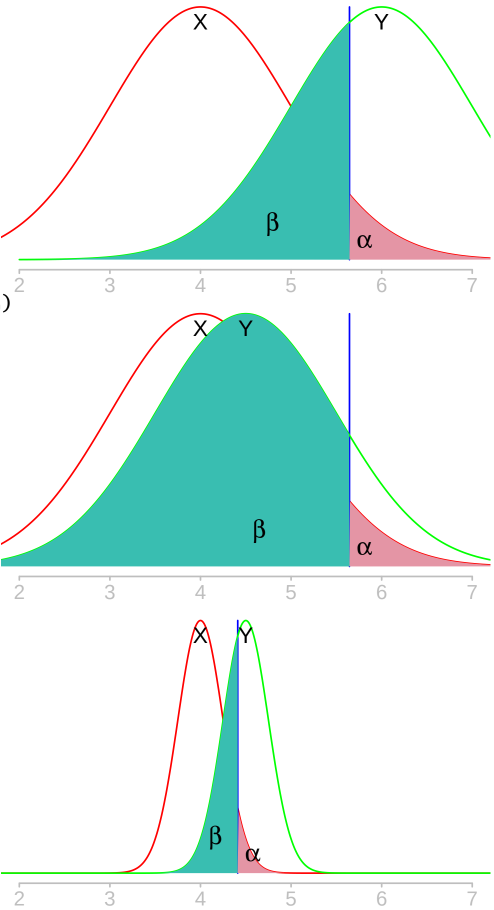


Figure 10.12: The impact of differences in mean and standard deviation on the overlap between two populations (α: probability of making a false positive error, and β: probability of making a false negative error). Github–Local

In the following code: h is the effect size (the ES.h function, from the pwr package, calculates this from the estimated proportion of runs that failed before/after the hardware change), n is the number of runs, sig.level is the p-value significance level and power the statistical power. The argument that is not specified (it is not necessary to specify NULL, this is the default value), or is given a NULL value, is returned by the call.

The default value of the alternative parameter is "two.sided".

```
library("pwr")

pwr.2p.test(h=ES.h(before, before+diff), n=num_runs, sig.level=0.05, power=NULL)
pwr.2p.test(h=ES.h(before, before+diff), n=NULL,     sig.level=0.05, power=0.8)
```

Figure 10.13, upper plot, shows the power achieved (y-axis), if a given difference in faults experienced does occur (x-axis), the before proportions 0.05, 0.25 and 0.5 are plotted; the power is plotted for 10 and 50 runs.

The probability of a difference being detected from 10 runs is below 0.5 (i.e., less than 50% chance of detecting a difference at a p-value of 0.05 or better), unless a change of hardware has a large impact on the proportion of faults experienced.

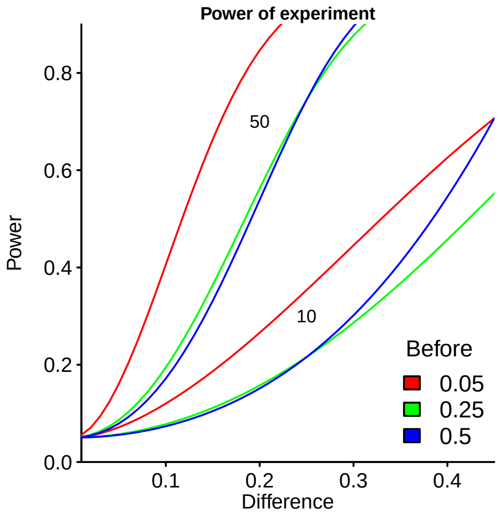
Figure 10.13, lower plot, shows the number of runs needed (y-axis), to have an 80% chance of detecting a given difference (x-axis) in proportion of faults experienced; the before proportions 0.05, 0.25 and 0.5 are plotted, at a significance of 0.05.

This lower plot can be used to find how much difference needs to be experienced for an experiment using 10 runs (per possible fault experience) to be likely to detect it. The failure of this experiment to detect any hardware configuration impact on number of known faults experienced, provides evidence that if any difference does exist, its impact is to add less than 50% or so to the proportion of intermittent fault experiences.

If the pwr package does not contain a function that calculates the power of the statistical test being considered, a Monte Carlo simulation can be used to perform a power calculation for the test being considered. The algorithm simulates an experiment, by obtaining samples from the population(s) that are thought to exist and performing the analysis on each sample, counting each success/failure to detect a difference.

The following code creates two populations and then compares two samples drawn from these populations. The user written function some_test_statistic compares two samples and returns the probability that an analysis of two samples will produce a given value; Github–statistics/boot-power.R contains an example that checks for a difference in mean value between samples drawn from two populations, see figure 10.14:

```
boot_power=function(pop_1, pop_2, sample_size, test_stat, alpha=0.05)
{
num_samples=5000 # Number of times to run the 'experiment'.
results=sapply(1:num_samples, function(X)
                {
                sample_1=sample(pop_1, size=sample_size, replace=TRUE)
                sample_2=sample(pop_2, size=sample_size, replace=TRUE)
                return(test_stat(sample_1, sample_2, alpha))
                })

return(sum(results<alpha)/num_samples) # fraction detected
}
```



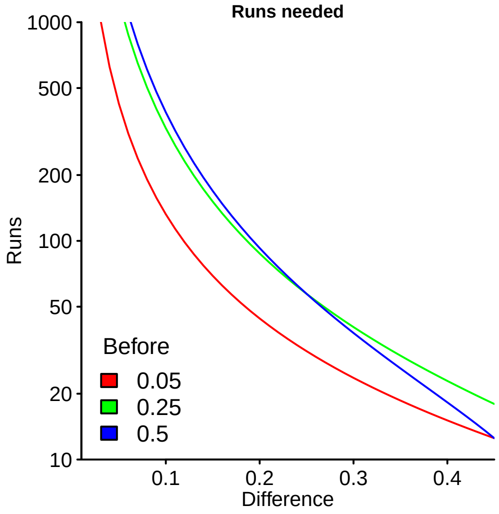


Figure 10.13: Power analysis (50 and 10 runs at various p-values) of detecting a difference between two runs having a binomial distribution (runs needed to achieve power=0.8 at various p-values). Github–Local

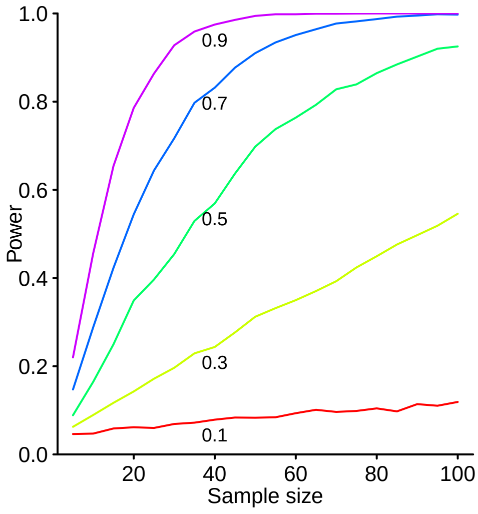


Figure 10.14: The statistical power of detecting that a difference exists between the mean values of samples of various sizes drawn from two populations; actual mean difference between samples adjacent to colored line. Github–Local

```
# Create two slightly different populations (which happen to be Normal here).
population_1=rnorm(100000, mean=0, sd=1)
population_2=rnorm(100000, mean=0+0.5, sd=1)

expected_sample_size=20 # The expected size of the sample to be collected
boot_power=function(population_1, population_2, expected_sample_size,
                              some_test_statistic, alpha=0.05)
```

Figure 10.14 shows the results of a Monte Carlo simulation that tests for a difference in the mean of two samples of various sizes, each drawn from a different population (see Github–statistics/response-power.R for the values calculated using an analytic solution, applicable for populations having a Normal distribution).

Obtaining good enough accuracy from a power analysis requires a good approximation of the likely characteristics of the sample obtained by an experiment. This information about the sample might be extracted from the results of related studies, a preliminary study or theory of the processes involved.

10.3 Describing a sample

A list of values can overwhelm readers with too much detail and techniques for compressing many values into a few, often just one value, are available.[vii] The few compressed values are known as *descriptive statistics*, and the following are some common sample descriptions:

- a point estimate of a central value and its variability, e.g., mean and standard deviation,
- an equation fitted to the sample data according to some condition, e.g., minimising mean squared error,
- quartiles, a cluster of measurements based on where values are relative to other values in the sample, e.g., a box-and-whiskers plot such as fig 8.20.

The mean and standard deviation are the two most commonly used descriptive statistics. It is incorrect to think that two distributions having the same mean and standard deviation will be very similar; see figure 10.15.

10.3.1 A central location

Perhaps the most widely used, single value summary of a sample, derives from the idea of a *middle* or *central* location.

- the mean, is perhaps the most commonly used central location; obtained by adding together the values, in a sample, and dividing by the number of values,
- the *median* is obtained by sorting the N values into numerical order and selecting the value of the $\frac{N+1}{2}$th element (if N is even the average of the middle two values is used),
- the *mode* is the value most likely to be sampled (R's mode function is unrelated to the statistical algorithm of that name, it returns the type or storage mode of an object). The modeest package contains functions for estimating various kinds of mode.

For symmetric distributions the values of the mean, median and mode are equal, while for asymmetric distributions, the three values can be very different.

When sample values are drawn from a unimodal distribution, the difference between the median and mean is less than or equal to $\sqrt{0.6}\sigma$, and for other non-unimodal distributions less than σ.

The difference between the median and mode is less than or equal to $\sqrt{3}\sigma$.

Unless the sample distribution is symmetric, it is not possible to sum multiple modes, e.g., cost estimates. For nonsymmetric distributions, adding underestimates the true value, e.g., for a Gamma distribution the mean is $k\theta$ and the mode is $(k-1)\theta$, where k and θ describe the Gamma distribution.

Figure 10.16 shows the distribution of execution times of the 1,000 input data sets from Chen et al.[336] If we are interested in an estimate of the execution time of a randomly

[vii] Plotting is a technique that can make use of all the values, and is the major focus of chapter 8.

chosen input data set, the median value, the point that equally divides the number of input data sets is the obvious choice. If we are interested in an estimate of the execution time most likely to be encountered, the value of the mode is the obvious choice.

Some distributions have such fat tails that the mean is infinite, e.g., the Cauchy distribution. In practice, the regularity with which very large values occur results in the mean value of a sample jumping around erratically, as new measurements are made. A distribution that does not have a finite mean may still have a median; the median is not affected by extreme values in the way the mean is, and any extreme values that do appear in a sample do not prevent the median converging to a fixed value.

The median absolute deviation is based around using the median as a robust estimation of variance; supported by the mad function.

The well-known algorithms for calculating the mean and standard deviation of a sample require that each value be independent of the others. When a sequence of values is serially correlated, i.e., the value of a measurement is related to the value of one or more immediately previous measurements, the calculated mean and standard deviation is biased. In the case of the mean, the uncertainty in its value grows for positive correlation, and decreases for negative correlation. Figure 10.17, upper plot, shows the fraction of this change for various sample sizes; it is based on an AR(1) model, where each value correlates with the immediately preceding value by an amount given in the legends on the right of the plot (see section 11.10 for a discussion of AR models). A positive correlation causes the ratio of the sample standard deviation, relative to the population standard deviation, to be underestimated, while a negative correlation causes it to be overestimated (figure 10.17, lower plot, shows the fraction of this change).

The sandwich package supports the calculation of various error measures that are caused by serial correlation (e.g., the lrvar function calculates the error in the long term mean of a series).

Circular data: Some measurements are made using a circular scale, with values that increase and wrap around from the maximum value to start again at the minimum value, e.g., angles take on values between 0 and 360.

The mean, if it exists, has a direction (or angle) and a length; figure 11.81 shows a calculated mean of values drawn from a circular distribution; see section 11.12.

Compositional data: The individual components of a sample of compositional data (i.e., data whose components always sum to a fixed value, such as percentages summing to 100%) are correlated, and the mean of each component cannot be calculated independently of the other components. The mean function in the compositions package calculates the mean of compositional data; see section.

Several methods of calculating the variance and standard deviation of compositional data have been proposed. The compositions package supports the mvar function, which calculates what is known as the *total variance* (or *generalized variance*), and the msd function which calculates the *metric standard deviation* (both return single values). The variation matrix includes information about the relationship between every pair of components, and is returned by the variation function; see Github–statistics/composite-variation.R for the variance calculation of the values plotted in figure 5.31.

10.3.2 Sensitivity of central location algorithms

Samples sometimes contain values that are noticeably different from the other values (e.g., much smaller or larger; which may or may not be the result of noise); the terms *outlier* or *influential observation* are used for such values. The percentage of sample values needed to cause a statistical estimator to produce an arbitrarily large (positive or negative) value is known as the *breakdown point*.

The breakdown point for the mean is proportional to $\frac{1}{N}$, i.e., no matter how many observations are made, it only takes one extreme value to produce a completely spurious result for the mean; the mean has the smallest breakdown point it is possible to have.

At the other end of the scale, the median has a breakdown point of 0.5 (i.e., half of the measurements can have extreme value without affecting the result value) and for this reason the median is often recommended, over the mean, when measurements values are known to be very noisy. However, the median cannot be recommended for universal use because there are situations where it does not perform as well as the mean. For

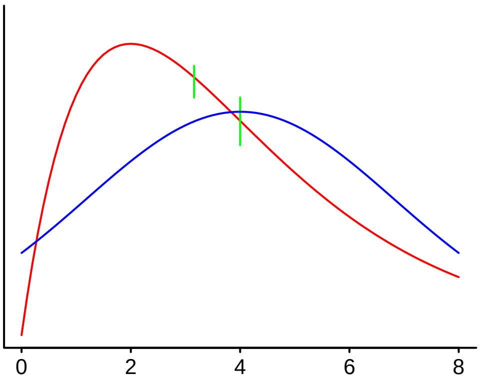


Figure 10.15: A Normal distribution with mean=4 and variance=8 and a Chi-squared distribution with four degrees of freedom having the same mean and variance (the vertical lines are at the distributions' median value). Github–Local

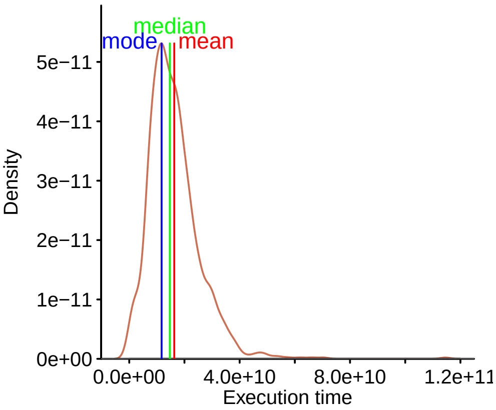


Figure 10.16: Density plot of execution time of 1,000 input data sets, with lines marking the mean, median and mode. Data kindly supplied by Chen.[336] Github–Local

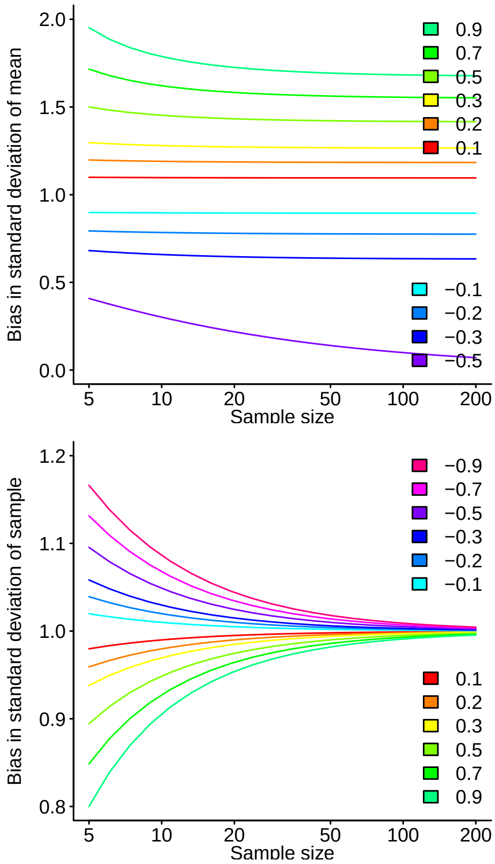


Figure 10.17: Impact of serial correlation, AR(1) in this example, on the calculated mean (upper) and standard deviation (lower) of a sample (the legends specify the amount of serial correlation). Github–Local

instance, when values are drawn from a discrete distribution whose mean is roughly halfway between measurable points, and the sample includes duplicate values, then most samples will have a median value slightly larger/smaller than the actual mean, i.e., the median is not evenly distributed across possible values in the way the mean is likely to be distributed; see figure 10.18.

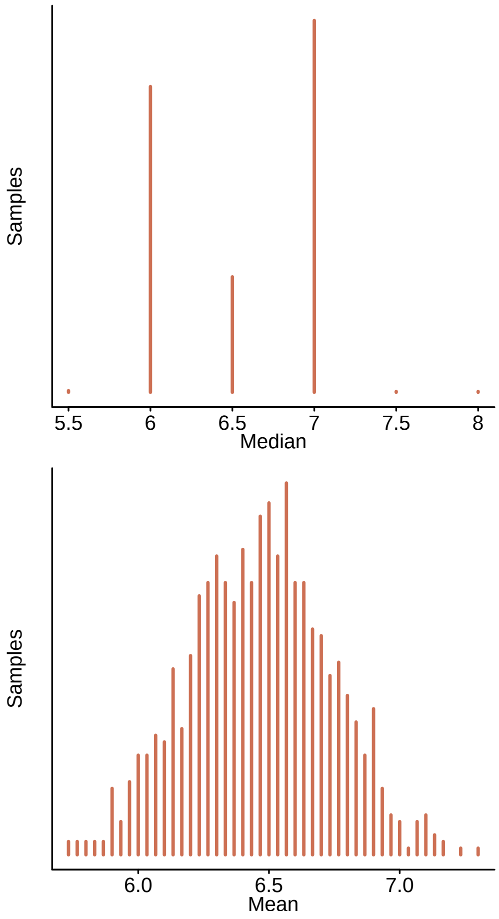


Figure 10.18: Number of sample median (upper) and mean (lower) values for 1,000 samples drawn from a binomial distribution. Github–Local

The probability of an outlier occurring depends on the reliability of the measurement process and the characteristics of the population being sampled. The following two techniques are robust in the presence of extreme values in a sample:

- *trimmed mean* removes a percentage of the largest and smallest values, before calculating the mean of the remaining values (it has been found that 20% is a good value for general use). The `mean` function includes a `trim` argument for specifying the percentage to be trimmed,
- *winsorized mean* replaces rather than remove values. The values of the lowest X% are replaced with the lowest value that is just not within the specified percentage, and the values of the highest X% are replaced with the highest value just not within this percentage; the Winsorized mean is calculated using the updated list of values. The `psych` package contains functions that calculate various quantities using the Winsorized mean.

The trimmed and winsorized means may produce biased results when applied to samples drawn from a population having an asymmetric distribution.

10.3.3 Geometric mean

The *geometric mean* of N values is:

$$Mean_g = \left(\prod_{i=1}^{N} X_i\right)^{\frac{1}{N}}$$

For instance, the geometric mean of 10, 100, 1000 is $(10 \times 100 \times 1000)^{\frac{1}{3}} \rightarrow 100$.

The geometric mean is preferred to the arithmetic mean when ranking ratios or normalised data (which is a kind of ratio), because it gives consistent results.

When one or more values, X_i, is negative or zero, calculating a geometric mean is a more complicated process.[754]

Consider the (invented) benchmark performance of the three systems in table 10.2. Treating a as the base performance, what is the relative performance improvement of b and c?

If the arithmetic mean is used, the performance ranking of b and c, relative to a, depends on whether the calculation used is a ratio of their means, or the mean of their ratios. The fourth column lists the mean of the values in the second and third column of the corresponding row, and the fifth column lists the ratio of these mean values (relative to a). The individual benchmark ratios for a and b are: $\frac{2}{1}$ and $\frac{105}{100}$, and for a and c: $\frac{3}{1}$ and $\frac{103}{100}$. The mean of these ratios is listed in the sixth column. Comparing columns five and six shows that the ranking of b and c depends on the method of calculating the ratios; also see table 13.1.

system	**integer**	**float**	**arithmetic mean**	**ratio of means**	**mean of ratios**	**geometric mean**
a	1	100	50.5			10
b	2	105	53.5	$\frac{53.5}{50.5} \rightarrow 1.0594$	`mean(2/1+105/100) -> 3.05`	14.49
c	3	103	53	$\frac{53}{50.5} \rightarrow 1.0495$	`mean(3/1+103/100) -> 4.03`	17.58

Table 10.2: Invented integer/float benchmark performance measurements of three systems and various methods of calculating relative performance. The relative performance of b and c depends on which mean is used.

If the geometric mean is used, the relative order of the final ratio is not order dependent.

Sometimes the arithmetic and geometric means produce the same benchmark rankings, e.g., a benchmark[552] of eight Intel IA32 processors used the arithmetic mean of ratios to compare results, the results from using the geometric means was not large enough to affect the relative ranking of processors for a given performance characteristic (see Github–benchmark/powervperfasplos2011.R).

The Geometric mean might be used when values cover several orders of magnitude, e.g., a geometric or logarithmic series (such as: 2, 4, 8, 16, 32, 64)

Methods for calculating the geometric mean include the expression `exp(mean(log(x)))`, and the `geometric.mean` function in the `psych` package.

10.3.4 Harmonic mean

The *harmonic mean* is used to find the "average" of a list of ratios or proportions; it is defined as:

$$Mean_h = \frac{N}{\sum_{i=1}^{N} \frac{1}{X_i}}$$

for instance, the harmonic mean of 1, 2, 3, 4, 5 is: $\frac{5}{\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}} \to 2.189781$

When there are two values the formula becomes:

$Mean_h = 2\frac{x \cdot y}{x+y}$, which has the same form as the F_1 score, or F-measure, used in information retrieval to combine the precision and recall:

$$F_1 = (1+\beta^2)\frac{precision \cdot recall}{\beta^2 precision + recall}$$

Two methods of calculating the harmonic mean are: `1/mean(1/x)`, and the `harmonic.mean` function in the `psych` package.

10.3.5 Contaminated distributions

A common assumption that often goes unquestioned, is that a sample, or the error present in the measurements it contains, is best described by a Normal distribution. Textbooks are filled with techniques that only exhibit the cited desirable attributes, when the Normality assumption holds. There is also the lure of analytic solutions to a problem, which again may only apply when the Normality assumption holds.

Even when sample values appear to be drawn from a Normal distribution, a small percentage of contaminated values can have a dramatic effect on the value returned by a statistical algorithm.

The Contaminated Normal distribution is a mixture of values drawn from two Normal distributions, both having the same mean, but with 10% of the values drawn from a distribution whose standard deviation is five times greater than the other. Figure 10.19 shows the kernel density of two samples, one containing 10,000 values drawn from a Normal distribution and the other containing 10,000 values from a Contaminated Normal distribution; visually they seem very similar (see Github–statistics/contam-norm.R to learn the color used to plot each sample).

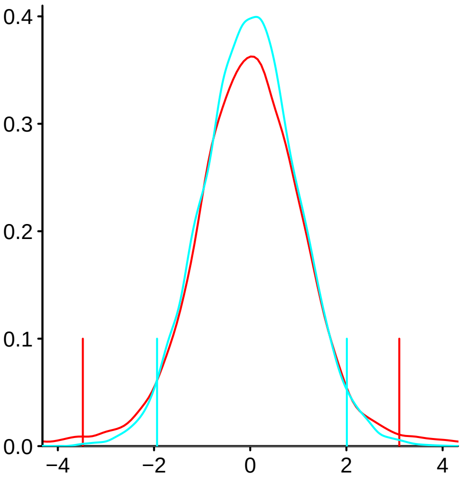


Figure 10.19: Density plot of two samples; samples either drawn from a Normal distribution or a Contaminated Normal distribution (i.e., values drawn from two normal distributions, with 10% of values drawn from a distribution having a standard deviation five times greater than the other); the lines bounding the 95% quartile identify the color used for each plot. Github–Local

This Contaminated Normal distribution has a standard deviation that is more than three times greater than the Normal distribution from which 90% of the values are drawn. This illustrates that a Normal distribution contamination by just 10% of values from another distribution can appear to be Normal, but have very different descriptive statistics.

A number of tests are available for estimating whether sample values have been drawn from a Normal distribution. The Shapiro-Wilk test (the `shapiro.test` function), the Kolmogorov-Smirnov Test (the `ks.test` function)[viii], and the Anderson-Darling test are common encountered. A comparison of four normality tests[1544] found the Shapiro-Wilk test to be the most powerful normality test; see fig 9.10.

When a data set contains very few values, even the Shapiro-Wilk test may fail to determine (e.g., p-value < 0.05) that sample values are not drawn from a Normal distribution. In the case of the Contaminated Normal distribution, samples containing only 10 values are considered to have a Normal distribution in around 30% of cases (i.e., p-value > 0.05), with the percentage dropping to 10% for samples containing 20 values.

Wilcox[1938] provides an analysis of potential problems that outliers, skewed distributions, and fat tails can cause.

[viii] Both included in R's base system.

10.3.6 Compositional data

Compositional data is an aggregate of components, each contributing a portion of the total, i.e., ideally summing to 1 or 100%. The requirement of a fixed total creates a correlation between the components, i.e., if one of component increases, one or more of the others has to decrease correspondingly. Failure to take this correlation between variables into account can analysis results having surprising characteristics, e.g., being unrealistic.

The theory needed to underpin techniques for handling compositional data is all very new, and many issues are still unresolved.

The compositions package supports the analysis of compositional data, and offers four approaches to the analysis of data (based on the geometries of the sample space). The compositional mapping functions are:

- aplus: the total amount matters, but amounts are compared relatively, e.g., the difference between 1 and 2 is treated the same as the difference between 100 and 200,
- rplus: the total amount matters, and amounts are compared absolutely, e.g., the difference between 1 and 2 is treated the same as the difference between 100 and 101,
- acomp: the total amount is constant, but amounts are compared relatively, e.g., the difference between 1 and 2 is treated the same as the difference between 100 and 200,
- rcomp: the total amount is constant, and amounts are compared absolutely, e.g., the difference between 1 and 2 is treated the same as the difference between 100 and 101.

Figure 5.31 shows the proportion of development time spent in the design, coding and testing phases of 39 applications. Which compositional mapping function is appropriate for this data? The measurements are hours spent in each phase (see Github–statistics/composite-variation.R), and from a project duration perspective a time difference of 1-hour is an absolute difference. When the percentage of total time spent in each phase is of interest, the rcomp function applies; when the absolute time is of interest, the rplus function applies.

```
library("compositions")

percent_phase=rcomp(est, parts=c("Design_Phase", "Code_Phase", "Test_Phase"))
hours_phase=rplus(est, parts=c("Design_Phase", "Code_Phase", "Test_Phase"))

mean(percent_phase)
mean(hours_phase)
```

The dist function can be used to calculate a distance between two compositional values. One use for a distance value is using the bootstrap to estimate the likelihood of a given difference between two mean values; see Github–projects/composite-mean-diff.R and section 10.5.2.

10.3.7 Meta-Analysis

Meta-analysis is the process of combining quantitative evidence from multiple studies to create more accurate estimates of the characteristics studied.

If descriptive statistics of each sample is the only information available, the mean and standard deviation can be pooled (creating a weighted single, combined value). The calculation is as follows (it assumes that each sample is independent of other samples; at the time of writing, no built-in functions are provided in R's base system):

```
pooled_mean=function(df)
{
return(sum(df$s_n*df$s_mean)/sum(df$s_mean))
}

pooled_sd=function(df)
{
return(sqrt(sum(df$s_sd^2*(df$s_n-1))/sum(df$s_n-1)))
}

studies=data.frame(s_n=c(5, 10, 20),
                   s_mean=c(30, 31, 32),
                   s_sd=c(5, 4, 3))
```

```
pooled_mean(studies)
pooled_sd(studies)
```

Medical and social science experiments often measure one or more characteristics of a system before/after an event (e.g., a drug or social program). Various meta-analysis techniques have been created to deal with this kind of before/after study; the `meta` package contains support for this analysis. In software engineering, replicating studies of this kind is not (yet) a common occurrence.

A study by Sabherwal, Jeyaraj and Chowa[1600] performed a meta-analysis of studies of the determinants of success of information systems projects, based on 612 findings from 121 studies published between 1980 and 2004.

The *file drawer problem* is the situation where the results from a study fail to reach the level of statistical significance needed for the work to be accepted for publication, i.e., a meta-analysis may be biased because the published results do not include studies with poor statistical significance (these results are sitting a file draws).[587]

A study by Bem[166] investigated *premonition*, i.e., a persons' ability to predict future events. In nine experiments subjects were asked to guess which of several stimuli would be randomly selected, after their response has been recorded. Experiment 1 involved 50 men and 50 women, who saw a screen containing two images of a curtain and were asked to select one of the curtain images. After a subject selected one curtain image, a picture of either a brick wall or of something else was revealed; the something else picture was either explicitly erotic or neutral. Each subject completed 36 trials. The sequencing of pictures, and their left/right position was randomly selected.

The results found that 53% of subjects selected the curtain image revealing an erotic image at a rate greater than chance (i.e., 50%); subject success rate for the neutral image was 47% (no significant subject sex difference was found). While bootstrap test shows that neither of these percentages occur less than 5% of the time (see Github–statistics/FeelingFuture.R), the binomial test used by the author found a statistically significant difference (the statistical analysis performed was as good as, or better, than that seen in most software engineering papers).

One solution to the file drawn problem is to preregister studies. Here, before collecting any data, researchers submit a description of the study and the data analysis techniques they plan to use; this information is kept confidential until the study is completed. Preregistration reduces the ability of researchers to engage in data dredging. One study[1049] found significant differences in 12 of the 15 meta-analysis studies analysed, compared using only published papers and then including preregistered studies.

10.4 Statistical error

The outputs from applying a statistical technique generally includes probabilities, and it is the responsibility of the person doing the analysis to decide the cut-off probability below/above which an event is considered to have/have not occurred.

The two kinds of statistical error that can be made are:

- treating a hypothesis as true when it is actually false; the statistical term is making a *Type I* error, but *false positive* is more commonly used, and expressed in mathematics: $P(\mathit{Type\ I\ error}) = P(\mathit{Reject}\ H_0|H_0\ \mathit{true})$,
- treating a hypothesis as false when it is actually true; the statistical term is making a *Type II* error, but *false negative* is more commonly used, and expressed in mathematics: $P(\mathit{Type\ II\ error}) = P(\mathit{Do\ not\ reject}\ H_0|H_A\ \mathit{true})$, where H_A is an alternative hypothesis.

		Decision made	
		Reject H	Fail to reject H
Actual	H true	Type I error	Correct
	H false	Correct	Type II error

Table 10.3: The four states available in hypothesis testing and their outcomes.

The practical consequences of a statistical error depend on who is affected by the outcome of the decision made. For instance, consider the consequences of a manager's decision on whether to invest more time and money testing the reliability of a software system. An incorrect decision can result in more than losing the original investment (e.g., losing market share to a competitor); the bearer of any loss depends on the actual situation and the decision made, as table 10.4 illustrates:

		Decision made	
		Finish testing	Do more testing
Actual	More testing needed	Customer loss	Ok
	Testing is sufficient	Ok	Vendor loss

Table 10.4: Finish/do more testing decision and outcome based on who incurs any loss.

10.4.1 Hypothesis testing

A hypothesis is an unverified explanation of why something is the way it is. Hypothesis testing is the process of collecting and evaluating evidence that may, or may not, be consistent with the hypothesis, i.e., positive and negative testing.[ix] Once enough evidence consistent with the hypothesis has been collected, people may feel confident enough to start referring to it as a theory or law.[446]

The most commonly used statistical hypothesis testing technique is based on what is known as the *null hypothesis*,[x] which works as follows:

- a hypothesis, H, having testable prediction(s) is stated,
- an experiment to test the prediction(s) is performed, producing data D,
- assuming the hypothesis is true, the probability of obtaining the data produced by the experiment is calculated. The calculation made is: $P(D|H)$; that is the probability of obtaining the data D, assuming that the hypothesis H is true.

  If the calculated probability is less than or equal to some prespecified value, the hypothesis is rejected, otherwise it is said that *the null hypothesis has not been rejected* (i.e., the result of the experiment is not conclusive evidence that the null hypothesis is true).

Expressed in code, the null hypothesis testing algorithm is as follows:

```
void null_hypothesis_test(void *result_data, float p_value)
{
// H is set by reality, only accessed by running experiments
if (probability_of_seeing_data_when_H_true(result_data) < p_value ||
     !H)
   printf("Willing to assume that H is false\n");
else
   printf("H might be true\n");
}

null_hypothesis_test(run_experiment(), 0.05);
```

A test statistic is said to be *statistically significant*, when it allows the null hypothesis to be rejected. The phrase "statistically significant" is often shortened to just "significant", a word whose common usage meaning is very different from its statistical one; this shortened usage is likely to be misconstrued when the audience is unaware that the statistical definition is being used, and treating the word as-if it is being used in its everyday meaning sense.

Statistical significance does not mean the pattern found by the analysis has any practical significance, i.e., the magnitude of the pattern detected may be so small as to make it useless for practical applications.

Running one experiment that produces a (statistically) surprisingly high/low p-value is a step in the process of increasing peoples' confidence that a hypothesis is true/false.

Replication of the results (i.e., repeating the experiment and obtaining similar measurements) provides evidence that the first experiment was not a chance effect; another boost

[ix] Gigerenzer[669] discusses how people make decisions in an uncertain environment.

[x] As the market leader in hypothesis testing techniques, over many decades, this technique attracts regular criticism.[371] The criticism is invariably founded on widespread misuse of the null hypothesis ritual; misuse is the fate of all widely used techniques.

in confidence. Replication by others, who independently set up and run an experiment, is the ideal replication (it reduces the possibility that unknown effects specific to a person or group influenced the outcome); an even larger boost in confidence.

There is a great deal of confusion surrounding how the results from a null hypothesis test should be interpreted. Studies have found[671] that people (incorrectly) think that one or more of the following statements apply:

- *Replication fallacy*: The level of significance measures the confidence that the results of an experiment would be repeatable under the conditions described. This is equivalent to saying: $P(D|H) == 1 - P(D)$, and would apply if the hypothesis was indeed true,
- the significance level represents the probability of the null hypothesis being true. This is equivalent to saying: $P(D|H) == P(H|D)$.

The Bayesian approach to hypothesis testing is growing in popularity and works as follows:

- two hypotheses, H_1 and H_2, having testable prediction(s) are stated (the second hypothesis may just be that H_1 is false),
- a non-zero probability is stated for the hypotheses being true, $P(H_1)$ and $P(H_2)$, known as the *prior* probabilities,
- an experiment to test the prediction(s) is performed (producing data D),
- the previously estimated probabilities, that H_1 and H_2 are true, is updated. The calculation uses Bayes theorem, which for H_1 is:

$$P(H_1|D) = \frac{P(H_1)P(D|H_1)}{P(H_1)P(D|H_1)+P(H_2)P(D|H_2)}$$

The updated prior probability, on the basis of the experimental data, is known as the *posterior probability* of the hypothesis being true.

10.4.2 p-value

In a randomized experiment, the *p-value* is the probability that random variation alone produces a test statistic as extreme, or more extreme, than the one observed.

The p-value for each coefficient of a fitted regression model (the subject of chapter 11) is a test of the hypothesis that the coefficient is zero, i.e., there is no association. When the actual value of a coefficient is close to zero, the reported p-value may be spurious. One solution is to rotate the axes, which will have the effect of increasing the value of the coefficient and removing this artefact from the p-value calculation (for this data).

In a commercial environment, the choice of p-value should be treated as an input parameter to a risk assessment comparing the costs and benefits of all envisioned possibilities.

In many social sciences, the probability of the null hypothesis being rejected is required to be less than 0.05 (i.e., 5%, or slightly less than 2σ),[xi] for a result to be considered worth publishing, while in civil engineering, a paper describing a new building technique that created structures having a 1-in-20 chance of collapsing would not be considered acceptable. High energy physics requires a p-value below $5\sigma \rightarrow 5.7 \cdot 10^{-7}$, for the discovery of a new particle to be accepted.

As sample size increases, p-values will always become smaller. For instance, if some aspect of flipping a coin very slightly favours heads, given enough coin flips a sufficiently small p-value, for the hypothesis that the coin is not a fair one, will be obtained. There is no procedure for adjusting p-values for hypothesis testing using very large amounts of data.

When lots of measurement data, covering many variables, is available it is possible to go on a fishing expedition, looking for relationships between variables.[1570] The probability of finding one significant result, when comparing n pairs of variables, using a p-value of 0.05, is $1-(1-0.05)^n$ (which is 0.4, when $n = 10$). When multiple comparisons are made, the base p-value needs to be adjusted to take account of the increased probability of noise being treated as a signal.

Perhaps the most common technique is the *Bonferroni correction*, which divides the base p-value by the number of tests performed. In the above example, the base p-value would

[xi] Journals with high impact factors can be more choosy, and some specify a p-value of 0.01.

be adjusted from 0.05 to $\frac{0.05}{10} \rightarrow 0.005$, to account for the possibility of each of the ten tests matching.

The `p.adjust` function supports p-value adjustment using a variety of different techniques.

Researchers know their work only has a chance of being accepted for publication, if the reported results have p-values below a journal's cut-off value. Given the use of published paper counts as a measure of academic performance, there is an incentive for researchers to run many slightly different experiments[1985] to find a combination that produces a sufficiently low p-value, that the work can be written up and submitted for publication[939] (a process known as *p-hacking*).[xii] One consequence of only publishing papers containing studies achieving a minimum p-value, is that many results are likely to be false (while a theoretical analysis suggests most are false,[883] an empirical analysis suggests around 14% of false positives for medical research[899]).

A study by Head, Holman, Lanfear, Kahn and Jennions[788] investigated the distribution of p-values appearing in the results section of Open Access papers in the PubMed database. Figure 10.20 shows the number of papers reporting a p-value equal to a given value, with fitted segmented regression model (four segments were specified, but the segment boundaries were selected by the fitting process).

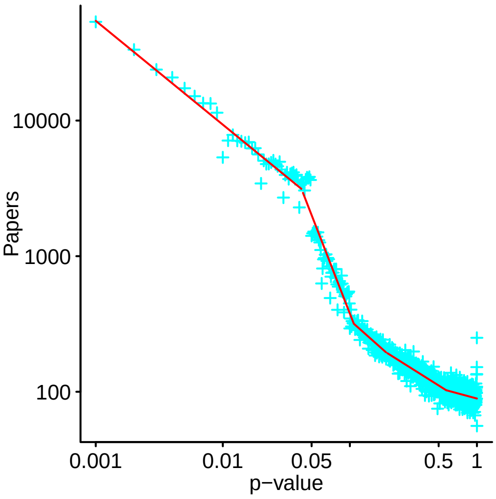


Figure 10.20: Number of papers reporting a p-value equal to a given value; lines are a fitted segmented regression model (four segments were specified). Data from Head et al.[788] Github–Local

10.4.3 Confidence intervals

Many statistical techniques return a single number, a point value. What makes this number so special, would a value close to this number be almost as good an answer? If an extra measurement was added to the sample, how likely is it that the original number would dramatically change; what if one measurement were excluded from the sample, how much would that change the answer?

A *confidence interval* is an upper and lower bound on the numeric point value(s) returned by statistical technique. A common choice is the 95% confidence bound, the default value used by many R packages.

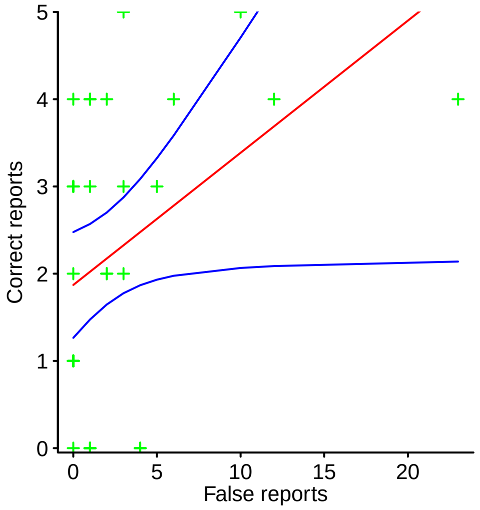


Figure 10.21: Regression model (red line; pvalue=0.02) fitted to the number of correct/false security code review reports made by 30 professionals; blue lines are 95% confidence intervals. Data from Edmundson et al.[523] Github–Local

Numeric confidence intervals can be mapped into visual form by adding them to a plot. Figure 10.21 illustrates how confidence intervals provide an easier to digest insight into the uncertainty of a fitted regression model, compared to the single number that is the p-value. The red line shows a fitted regression model, whose predictor has a p-value of 0.02; the 95% confidence intervals in blue, showing how wide a range of lines could be said to fit the sample almost as well (i.e., any straight line bounded by the blue lines).

A confidence interval is a random variable, it depends on the sample drawn. If many 95% confidence intervals are obtained (one from each of many samples), the true fitted model is expected to be included in this set of intervals 95% of the time (it is a common mistake to think that the confidence interval of one sample has this property). The probability that the next sample will be within the 95% confidence interval of the current sample, for a Normal distribution, is 84% or around 5 out of 6.[413]

A closed form formula for calculating confidence intervals is only known for a few cases, e.g., the mean of samples drawn from a Normal distribution; for a Binomial distribution a variety of different approximations have been proposed.[1470]

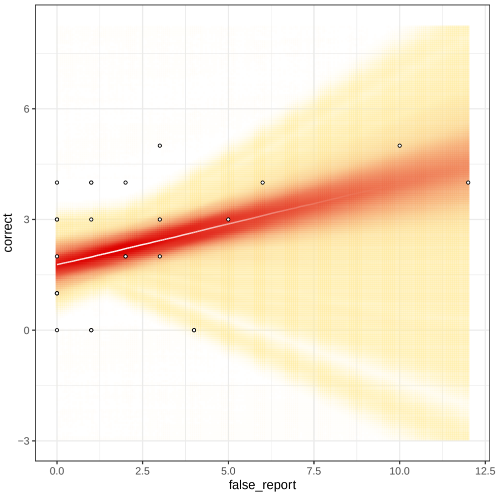


Figure 10.22: Bootstrapped regression lines fitted to random samples of the number of correct/false security code review reports made by 30 professionals. Data from Edmundson et al.[523] Github–Local

Built-in support for calculating confidence intervals, in R packages, is sporadic. Monte Carlo simulation can be used to calculate a confidence interval from the sample, e.g., the bootstrap. This approach has the advantage that it is not necessary to assume that sample values are drawn from any particular distribution. Figure 10.22 was created by fitting many models, via bootstrapping, and using color to indicate density of fitted regression lines.

10.4.4 The bootstrap

The bootstrap is a general technique for answering questions about uncertainties in the estimate of a statistic calculated from a sample, e.g., calculating a confidence interval or standard error.[813] Bootstrap techniques operate on a sample drawn from a population, and cannot extract information about the population that is not contained in the sample,

[xii] Which commercial company would not be willing to add warts to their software to keep an important customer happy?

e.g., if the population contains reds and greens, and a sample only contains reds, then the bootstrap will not provide any information about the greens.

The term *bootstrapping* denotes the process by which a computer starts itself from an off-state. In statistics, it is used to denote a process where new samples are created from an existing sample; the term *resampling* is sometimes used.

The bootstrap procedure often starts by assuming there is no difference, in some characteristic, between samples; it then calculates the likelihood of two samples having the characteristic they are measured to have. The assumption of no difference requires that the items in both samples be *exchangeable*. Deciding which items, if any, in a sample are exchangeable is a crucial aspect of using the bootstrap to answer questions about samples.

Individual time series measurements contain serial correlations. The block bootstrap is one technique for applying bootstrap techniques to time series data. The `tsboot` function, in the `boot` package, supports the bootstrapping of time series data.

Estimating the confidence interval for the mean value of a sample is a good example of the basic bootstrap algorithm; the steps involved are as follows:

- create a sample by randomly drawing items from the original sample. Usually the items are selected with replacement (i.e., an item can be selected multiple times). When items are selected without replacement (i.e., can only be selected once), the term *jacknife* is used,
- calculate the mean value of the created sample,
- iterate the create/calculate cycle, say, 5,000 times,
- analyze the 5,000 mean values, to obtain the lowest and highest 2.5%. The 95% confidence interval for the mean of the original sample is calculated from this lowest/highest band (several algorithms, giving slightly different answers, are available).

The `boot` package supports common bootstrap operations, including the `boot.ci` function for obtaining a confidence interval from a bootstrap sample.

The distribution of the sample from which the bootstrap algorithm draws values is known as the *empirical distribution*.

The *bootstrap distribution* contains m^n possible samples, when sampling with replacement from m possible items to create samples containing n items; when the order of items does not matter, there are $\binom{2m-1}{n}$ possible samples (a much smaller number).

The same bootstrap procedure can be applied to obtain confidence intervals on a wide range of metrics. Figure 10.23 shows confidence intervals for the kernel density plotted in figure 8.14, and was produced by the `sm.density` function, in the `sm` package, using the following code:

```
library("sm")

res_sample=sample(cint$Result, size=1000) # generate 1000 samples

sm.density(res_sample, h=4, col=point_col, display="se", rugplot=FALSE,
        ylim=c(0, 0.03),
        xlab="SPECint Result", ylab="Density\n")
```

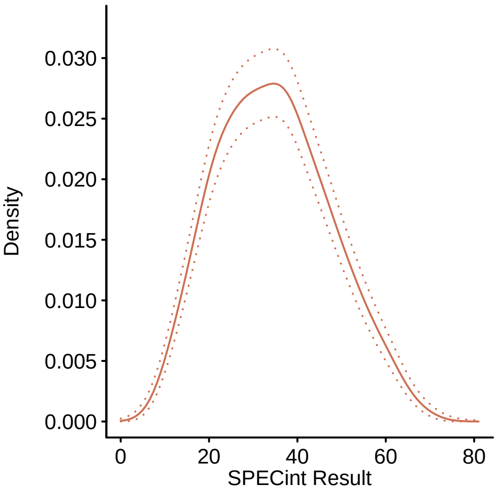


Figure 10.23: Kernel density plot, with 95% confidence interval, of the number of computers having the same SPECint result. Data from SPEC.[1720] Github–Local

The importance of using the appropriate sample size, when using the bootstrap, is illustrated by the analysis of the data from a study by Davis, Moyer, Kazerouni and Lee,[437] which investigated the use of regular expressions in eight languages; the sample size varied between languages. The regex library provided by each language supports different matching functionality, and to handle this the researchers mapped regexs found in each language's source code to a common representation. This mapping makes it possible to assumes that regexs in their common representation form are interchangeable.

Table 10.5 shows, for each language, the mean length of regular expressions, sample size, and the bootstrap probability that the mean observed is less than the bootstrapped means. While Rust has the longest regex mean length, its sample size is relatively small, and the bootstrap finds that it is not possible to rule out that possibility that the mean length observed is not unusual (i.e., 8.8% of generated samples had a mean greater than 39.9). Javascript and Java have, respectively, the second and third longest mean lengths, and their larger sample sizes reduces the uncertainty in the expected mean length; a mean regex length as large as the ones seen is very unlikely to be encountered (i.e., none appeared in the generated samples).

Language	mean	sample_size	Probability
rust	39.9	2005.0	8.1
go	30.2	21882.0	99.7
python	32.4	43486.0	78.9
php	27.7	43809.0	100.0
perl	23.7	141393.0	100.0
javascript	38.8	149479.0	0.0
ruby	33.7	151898.0	16.0
java	37.6	165859.0	0.0

Table 10.5: Mean length of sample of regular expressions in languages and bootstrapped probability of occurrence. Data from Davis et al.[437] Github–Local

10.4.5 Permutation tests

For small sample sizes, many computers are fast enough for it to be practical to calculate a statistic (e.g., the mean) for all possible permutations of items in a sample. This kind of test is known as a *permutation test*. Permutation tests do not have any preconditions on the distribution of the sample, other than it be representative of the population, and they return an exact answer.

Some techniques designed for manual implementation (e.g., Student's t-test) are approximations to the exact answer returned by a permutation test.

The coin package contains infrastructure for creating permutation tests and functions that perform common tasks (the names of these functions are derived from the names of the tests designed for manual implementation, e.g., spearman_test and wilcox_test).

The following permutation test calculates the likelihood that the professional experience of the two samples of subjects appearing in figure 8.3 have different mean values:

```
library("coin")

# The default is alternative="two.sided",
# an option not currently listed in the Arguments section.
oneway_test(experience ~ as.factor(language), data=Perl_PHP, distribution="exact")
```

10.5 Comparing samples

The need to compare measurements, obtained from running experiments, kick started the development of statistics. The wide range of experimental designs (e.g., one/two/k samples, parametric/non-parametric and between/within subject), along with the need for practical manual solutions, resulted in the evolution of techniques designed to do a good job of handling each specific kind of comparison. This book assumes a computer is available to do the number crunching, and uses either regression (covered in chapter 11), or the bootstrap.[xiii]

Samples may be compared to check whether they are the same/different, in some sense, or by specifically testing whether one sample is greater than, or less than, the other:

- in a *two-sided* test (also known as a *two-tailed* or *non-directional* test) the samples are checked for being the same or different, where an increase or decrease in some attribute is considered a difference. Figure 10.24, upper plot, the percentage on each side is half the chosen p-value,
- in a *one-sided* test (also known as a *one-tailed* or *directional* test) the samples are checked for only one case, either an increase or a decrease in the measured attribute. Figure 10.24, lower plot, the percentage on the one side is the chosen p-value.

A commonly encountered null hypothesis, when comparing two samples, is that there is no difference between them. In many practical situations a difference is expected, or hoped, to exist, otherwise no effort would have been invested in obtaining the data needed to perform the analysis.

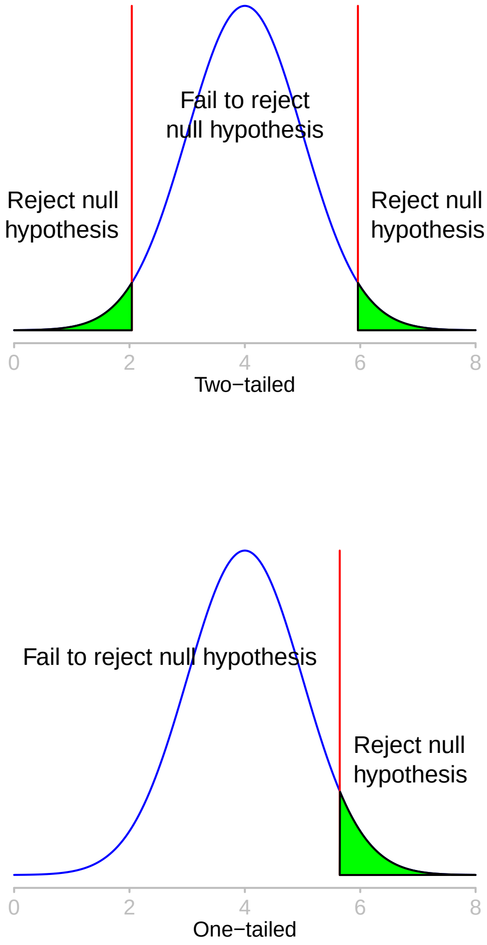


Figure 10.24: One and two-sided significance testing. Github–Local

[xiii] Other books tend to primarily cover the manual techniques: such as the t-test, which is a special case of multiple regression using an explanatory variable indicating group membership, and the Wilcoxon-Mann-Whitney test, which is essentially proportional odds ordinal logistic regression.

Experiments are often performed because a difference in one direction is of commercial interest. However, expecting or wanting a result that shows a difference in one direction is not sufficient justification for using a one-sided statistical test.

A one-sided test should only be used when the direction is already known, or when an effect in the non-predicted direction would be ignored. If an effect in a particular direction is expected, but an effect in the opposite direction would not be ignored (i.e., would be considered significant) a two-sided test should be used.

Some of the kinds of sample comparisons commonly made include:

- a level of confidence that sample values have been drawn from the same/different distribution (discussed in section 9.2.1),
- the difference, d_m, in the mean of two samples,
- the difference, d_v, in the variance of two samples,
- the correlation, C, between values, paired from two samples.

Correlated measurements: Many data analysis techniques assume that each measurement is independent of other measurements in the sample.

An experiment that measures the same subject before and after the intervention (i.e., a within-subjects design; a between-subjects design involves comparing different subjects) involves correlated data. One technique for handling this kind of correlated data is mixed-effects models, discussed in section 11.6.

Time dependent measurements may be correlated, with later measurements affected by earlier events that are not part of the benchmark (say). A correlation between successive measurements, where none should exist, either needs to be removed or taken into account during analysis. The Durban Watson test can be used to check for a correlation between successive measurements within each run. The `durbinWatsonTest` function, in the `car` package implements this test; see the discussion associated with figure 11.22.

Time series analysis deals with sequentially correlated data, see section 11.10.

10.5.1 Building regression models

Using regression modeling to analyse data may appear to be over-kill (it is used to analyse many of the datasets appearing in this book). When a computer is available to do the work, it makes sense to use the most powerful analysis techniques available that has the fewest preconditions; learning to apply the appropriate, less powerful, technique, often with stronger preconditions, is a waste time (unless you don't have access to a computer).

Techniques designed for manual implementation, such as Pearson correlation, Spearman correlation, t-test, Wilcoxon signed-rank test, etc., are all special cases of regression; for examples of the correspondence with regression, see Github–statistics/manual-tests.R. Manual implementation techniques for comparing two or more samples have been made obsolete by the bootstrap (covered in section 10.4.4), when a computer is available.

Regression provides a simple unified framework for dealing with many data analysis problems; it is possible to start with a simple model, and progressively add more features.

A study by Potanin, Damitio and Noble[1493] refactored the Java Development Kit collection so that it nolonger made use of incoming aliases (e.g., following the owner-as-dominator or owner-as-accessor encapsulation discipline). The DaCapo benchmark,[204] which contains 14 separate programs, was used to compare the performance of the original and refactored versions. The programs were each run 30 times, with measurements made during each of the last five iterations; this process was repeated five times, generating 25 measurements for each program for a total of 350 measurements.

The researchers claimed that their changes to the aliasing properties of the original code did not degrade performance. If the claim is true, the explanatory variable kind-of-refactoring, will have a trivial impact on the quality of the fitted regression model. The simplest model possible is based on the program name, and explains 99.9% of the variance (in this case the intercept is an unnecessary degree of freedom):

```
prog_mod=glm(performance ~ progname-1, data=dacapo_bench)
```

The fitted equation contains just the mean value of the runtime of each separate program, for all programs in the sample. The `summary` function lists the details of the fitted model as: Github–Local

```
Call:
glm(formula = performance ~ progname - 1, data = dacapo)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-4174.6   -205.0     -9.0    116.6   3946.5

Coefficients:
                   Estimate Std. Error  t value Pr(>|t|)
prognameavrora     22881.32      48.03  476.439  < 2e-16 ***
prognamebatik       2519.87      48.03   52.469  < 2e-16 ***
prognameeclipse    53660.53      48.03 1117.330  < 2e-16 ***
prognamefop          395.89      48.03    8.243 2.92e-16 ***
prognameh2         24100.39      48.03  501.823  < 2e-16 ***
prognamejython     15808.13      48.03  329.160  < 2e-16 ***
prognameluindex      708.00      48.03   14.742  < 2e-16 ***
prognamelusearch    7239.52      48.03  150.743  < 2e-16 ***
prognamepmd         4017.61      48.03   83.656  < 2e-16 ***
prognamesunflow    22788.81      48.03  474.513  < 2e-16 ***
prognametomcat      7672.11      48.03  159.750  < 2e-16 ***
prognametradebeans 27987.82      48.03  582.768  < 2e-16 ***
prognametradesoap  64888.58      48.03 1351.122  < 2e-16 ***
prognamexalan      26381.35      48.03  549.318  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 345970)

    Null deviance: 1.5873e+12  on 2100  degrees of freedom
Residual deviance: 7.2169e+08  on 2086  degrees of freedom
AIC: 32759

Number of Fisher Scoring iterations: 2
```

Adding kind-of-refactoring as an explanatory variable (see Github–regression/dacapo_progname.R for details), finds that it is not significant on its own, but an interaction exists between it and a few programs (primarily sunflow). The model:

```
prog_refact_mod=glm(performance ~ progname+progname:refact_kind,
                                                  data=dacapo_bench)
```

explains 99.92% of the variance. There are 12 program/refactoring interactions with p-values less than 0.05 (out of 84 possible interactions), with most of these changing the estimated mean performance by around 1% and one making 8% difference (i.e., sunflow; see Github–regression/dacapo_progname_refact.R).

Building a regression model has enabled us to confirm that, apart from a few, small, interactions the various refactorings of JDK did not change the DaCapo benchmark performance.

10.5.2 Comparing sample means

Comparing two samples, to check for a difference in their mean values, is perhaps the most common statistical test performed. The bootstrap is a general purpose technique for answering sample comparison questions; see section 10.4.4.

The nVidia GTX 970 is a popular graphics card, with many variations on the reference design being sold (during August 2016 there were 51 variants included in the 64,392 results for this card in the UserBenchmark.com database). Figure 10.25 shows the number of Reflection benchmark results reported for GTX 970 cards, from three third-party manufacturers.

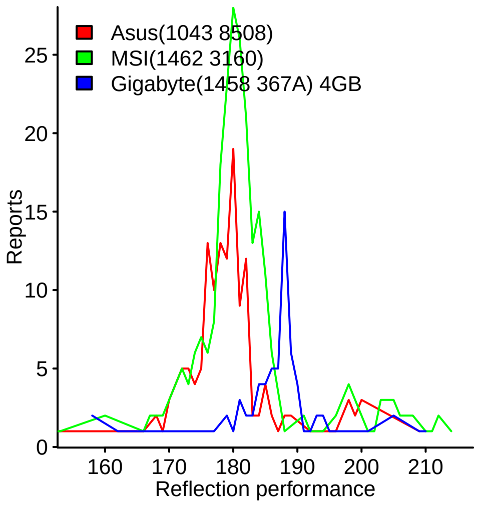


Figure 10.25: Number of Reflection benchmark results achieving a given score, reported for GTX 970 cards from three third-party manufacturers. Data extracted from UserBenchmark.com. Github–Local

The mean score of these Asus, MSI and Gigabyte cards are 176.2, 179 and 186.8 respectively. Are these differences more likely to be the result of random variation or by some real hardware/software difference?

The bootstrap can be used to answer this question, as follows.

Assume there is no difference in the mean performance of, say, MSI and Gigabyte on the Reflection benchmark. In this case the benchmark results (255 from MSI and 73 from Gigabyte) can be merged to form a sample of 328 results. Using this combined empirical sample perform the following:

- randomly select, with replacement, 328 items from the empirical sample,
- divide this new sample into two subsamples, randomly selecting one to contain 255 items and the other 73 items,
- find the mean of the two subsamples, subtract the two mean values and record the result,
- repeat this process R times,
- count how many of these bootstrapped differences in the mean are greater than the differences in the means of the two cards; no assumption is made about the direction of the difference, i.e., this is a two-sided test.

The following code uses the boot function, from the boot package, to implement the above algorithm, with the user provided function (mean_diff in this case) that is called for each randomly generated sample (see Github–group-compare/UserBenchmark_compare.R):

```
library("boot")

mean_diff=function(res, indices)
{
t=res[indices]
return(mean(t[1:num_MSI])-mean(t[(num_MSI+1):total_reps]))
}

MSI_refl=MSI_1462_3160$Reflection
Giga_refl=Gigabyte_1458_367A$Reflection

num_MSI=length(MSI_refl)      # Size of each sample
num_Giga=length(Giga_refl)
total_reps=num_MSI+num_Giga  # Total sample size

GTX_boot=boot(c(MSI_refl, Giga_refl), mean_diff, R = 4999) # bootstrap

refl_mean_diff=mean(MSI_refl)-mean(Giga_refl) # Difference in sample means
# Two-sided test
length(GTX_boot$t[abs(GTX_boot$t) >= abs(refl_mean_diff)]) # == E
```

The argument R specifies the number of resamples, with boot returning the result of calling mean_diff for each resample.

The likelihood of encountering a difference in mean values, as large as that seen in the MSI and Gigabyte performance (i.e., the p-value), is given by the equation: $\frac{E+1}{R+1}$, where: E is the number of cases where the bootstrap sample had a larger mean difference. The result varies around: $\frac{34+1}{4999+1} \to 0.007$ (the MSI/Asus the value is: $\frac{840+1}{4999+1} \to 0.17$).

If there were no difference in performance, a difference in mean value as large as that seen for MSI/Gigabyte is expected to occur 0.7% of the time. Based on a 5% cut-off, we can claim this percentage is so small that there is likely to be a real difference in performance. A mean difference at least as large as the MSI/Asus mean difference, is likely to occur 17% of the time, when there was no real difference in performance; a large enough percentage to infer that there is unlikely to be any difference in performance.

If a difference is thought likely to exist, the next question is the likely size of the difference, and the confidence intervals on this value. A bootstrap procedure can be used to answer these questions.

Once two samples are considered to be different, items within each sample can only be treated as exchangeable with other items within the corresponding sample. The two subsample now have to be selected from their respective empirical samples, as in the following code (see Github–group-compare/UserBenchmark_mdiff.R):

```
library("boot")

mean_diff=function(res, indices)
{
t=res[indices, ]
return(mean(t$refl[t$vendor == "Gigabyte"])- mean(t$refl[t$vendor == "MSI"]))
}

# Need to identify vendor used for each measurement.
MSI_refl=data.frame(vendor="MSI", refl=MSI_1462_3160$Reflection)
Giga_refl=data.frame(vendor="Gigabyte", refl=Gigabyte_1458_367A$Reflection)

MSI_Giga=rbind(MSI_refl, Giga_refl)

# Pass combined dataframe and specify identifying column
GTX_boot=boot(MSI_Giga, mean_diff, R = 4999, strata=MSI_Giga$vendor)
```

The boot.ci function calculates confidence intervals from the values returned by boot (in this case, the difference in mean values): Github–Local

```
> boot.ci(GTX_boot)
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 4999 bootstrap replicates

CALL :
boot.ci(boot.out = GTX_boot)

Intervals :
Level      Normal              Basic
95%   ( 4.426, 11.204 )   ( 4.196, 11.118 )

Level     Percentile            BCa
95%   ( 4.511, 11.433 )   ( 4.956, 12.042 )
Calculations and Intervals on Original Scale
> mean(GTX_boot$t)
[1] 7.813968
> sd(GTX_boot$t)
[1] 1.729132
```

Deciding if, and when, items in a sample are exchangeable can be non-trivial and requires an understanding of the problem domain.

A study by Gandomani, Wei and Binhamid[640] investigated the accuracy of software cost estimates, made using both expert judgement and Planning Poker, on 15 projects in one company, and both expert judgement and Wideband Delphi in 17 projects in another company; table 10.6 shows a subset.

Is there a difference in the estimates made using expert judgement and either of the other two techniques?

Project	Expert judgement	Planning Poker	Difference
P1	41	40	1
P4	60	56	4
P7	33	45	-12
P12	18	20	-2

Table 10.6: Effort estimates made using expert judgement and Planning Poker for several projects. Data from Gandomani et al.[640]

Each estimate is specific to one project, and it makes no sense to include estimates from other projects in the random selection process; estimates from different projects are not exchangeable. Possible ways of handing this include:

- treating each project as being exchangeable; the resampling could occur at the project level with both estimates for each selected project being used. This makes no sense, from the business perspective.
- randomly selecting from the set of estimates for each project. This possibility is not ruled out, from the business perspective, even though there are only two estimates for each project.

The following code randomly samples estimates for each project (see Github–group-compare/16.R):

```
mean_diff=function()
{
```

```
s_ind=rnorm(len_est_2) # random numbers centered on zero
# Randomly assign estimates to each group
expert=c(est_2$expert[s_ind < 0], est_2$planning.poker[s_ind >= 0])
# Sampling with replacement, so two sets of random numbers needed
s_ind=rnorm(len_est_2) # random numbers centered on zero
poker=c(est_2$expert[s_ind < 0], est_2$planning.poker[s_ind >= 0])
# The code for sampling without replacement
# poker=c(est_2$expert[s_ind >= 0], est_2$planning.poker[s_ind < 0])
return(mean(expert)-mean(poker))
}

est_mean_diff=abs(mean(est_2$expert)-mean(est_2$planning.poker))
len_est_2=nrow(est_2)

t=replicate(4999, mean_diff()) # Run the bootstrap

# What percentage of means are as large as the experiment?
100*length(which(abs(t) > est_mean_diff))/(1+length(t))
```

The p-value for a two-sided test between Expert and Planning Poker is 0.02 (see Github–group-compare/16.R), which suggests there is a difference, but does not provide any information about the direction of difference.

A study by Jørgensen and Carelius[938] asked companies to bid on a software development project.[xiv] In the first round of bidding 17 companies were given a one-page description of user needs and asked to supply a non-binding bid; in the second round the original 17 companies plus an additional 18 companies (who had not participated in the first round) were given an 11-page specification (developed based on feedback from the first round) and asked to submit firm-price bids.

What difference, if any, did participating in the first round make to the second bids, submitted by the initial 17 companies (call them the A companies) and how did these bids compare to those submitted by the second sample of 18 companies bidding for the first time (call them the B companies)?

Figure 10.26 shows density plots of the submitted bids. The mean values were: kr183,051[xv] for initial bid from A companies, kr277,730 for final bid from A companies and kr166,131 for single bid from B companies.

Are the items in each sample (the companies asked to submit a bid) exchangeable? Small companies have lower operating costs than large companies; it is unrealistic to consider bids from small/large companies to be exchangeable. The size of companies involved in bidding were classified as small (five or fewer developers), medium (between 6 and 49 developers) and large (50 or more developers).

The call to boot has to include information on how the data is stratified (i.e., split into different levels). The argument strata is used to pass a vector of integer values specifying the strata membership of the values present in the first argument. Everything else stays the same, with boot treating members of each strata as exchangeable when generating, new samples (see Github–group-compare/compare-bid.R):

```
bid_boot=boot(comp_bid$Bid, mean_diff, R = 4999,
                                 strata=as.factor(comp_bid$CompSize))
```

The p-value, for the hypothesis that the mean values are the same, is: $\frac{52+1}{4999+1} \rightarrow 0.01$, i.e., a difference this large is (statistically) surprising.

Jørgensen and Carelius proposed the hypothesis that the main factor controlling the size of the bids was the information contained in the project specification. I think this is rather idealistic, more practical considerations are discussed in section 5.2.

The intervals of a time series are, by their very nature, not exchangeable. Bootstrapping a time series requires its own distinct algorithm; the tsboot function handles the details.

Permutation tests: When the two samples contain only a few items, it is practical to generate and test all possible item permutations.

A study by Grant and Sackman[719] measured the time taken for subjects to write a program using either an online or offline computer interface (this experiment was run during the

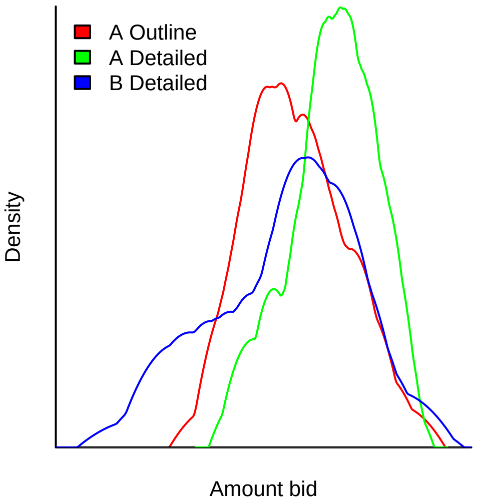


Figure 10.26: Density plots of project bids submitted by companies before/after seeing a requirements document. Data from Jørgensen et al.[938] Github–Local

[xiv] Four of the companies that submitted a bid were selected to independently implement the project.

[xv] The exchange rate was approximately 10 Norwegian Krone to one Euro.

1960s mainframe era). Given 12 subjects, split into two groups of six, how likely is the difference in mean time between the online/offline use cases?

This question is about the population of people who took part in the experiment, not a wider population. For this population there are `choose(12, 6)` ==924 possible subject combinations. The following is an excerpt of an implementation of a two-sided test (see Github–group-compare/GS-perm-diff.R):

```
subj_time=c(online$time, offline$time)  # Combine samples
subj_mean_diff=mean(online$time)-mean(offline$time)

# Exact permutation test
subj_nums =seq(1:total_subj)
# Generate all possible subject combinations
subj_perms=combn(subj_nums, subj_online)

mean_diff = function(x)
{
# Difference in mean of one combination of subjects
mean(subj_time[x]) - mean(subj_time[!(subj_nums %in% x)])
}

# Indexing by column iterates through every permutation
perm_res=apply(subj_perms, 2, mean_diff)

# p-value of two-sided test
sum(abs(perm_res) >= abs(subj_mean_diff)) / length(perm_res)
```

For the Algebra program, 272 of the possible groups, of subject combinations, had a difference in mean time greater than, or equal, to that of the empirical sample. Because all possibilities have been calculated, the p-value is exact: $\frac{272}{924} \to 0.2934$.

The `coin` package provides this kind of exact calculation for many of the traditional group comparison tests, e.g., the `wilcoxsign_test` function is the permutation test equivalent of the `wilcox.test` function (in the base library).

The bootstrap techniques used to answer questions about differences in the mean of two samples, can be generalised to a wide variety of comparison tests. A new comparison test can be implemented by replacing the `mean_diff` function used in the earlier examples (the requirement of exchangeability remains an integral requirement).

10.5.3 Comparing standard deviation

A study by Jørgensen and Moløkken[942] asked 19 professional developers to estimate the effort required to implement a task, along with an uncertainty estimate (i.e., minimum and maximum about the most likely value). Nine of these developers were explicitly instructed to compare the current task with similar projects they had worked on (they were also given a table that asked them to assess similarity within various percentage bands).

The visual appearance of the density plots, in figure 10.27, suggests that there is a difference in the standard deviation of the estimates in the two samples. A bootstrap test, of the difference in the standard deviations of the two samples, can be implemented by replacing the `mean_diff` function used in the previous section, by the function `sd_diff` as follows; see Github–group-compare/simula_04sd.R:

```
sd_diff=function(est, indices)
{
t=est[indices]
return(sd(t[1:num_A_est])-sd(t[(num_A_est+1):total_est]))
}
```

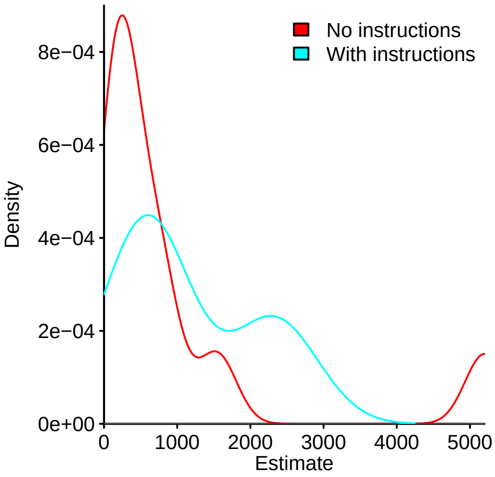


Figure 10.27: Density plot of task implementation estimates: with no instructions (red) and with instruction on what to do (blue). Data from Jørgensen el al.[942] Github–Local

The p-value, for the hypothesis that the standard deviations are the same, is: $\frac{2170+1}{4999+1} \to 0.43$, i.e., the difference is not that (statistically) surprising.

The `ansari_test` function, in the `coin` package, [xvi] performs an *Ansari-Bradley Test* (a two-sample permutation test for a difference in variance); see Github–group-compare/simula_04_var.R.

[xvi] The `ansari.test` function is included in R's base system.

10.5.4 Correlation

Correlation is a measure of linear association between variables, e.g., the extent to which one variable always increases/decreases when another variable increases/decreases. The range of correlation values is -1 (the variables change together, but in opposite directions) to 1 (the variables always change together), with zero denoting no correlation.

Correlation is related to regression, except that: it treats all variables equally (i.e., there are no response or explanatory variables), the correlation value is dimensionless and correlation is a linear relationship (i.e., there need not be any correlation between variables having a non-linear relationship, e.g., in the $y = x^2$ relationship, y can be predicted x, but there is zero correlation between them).

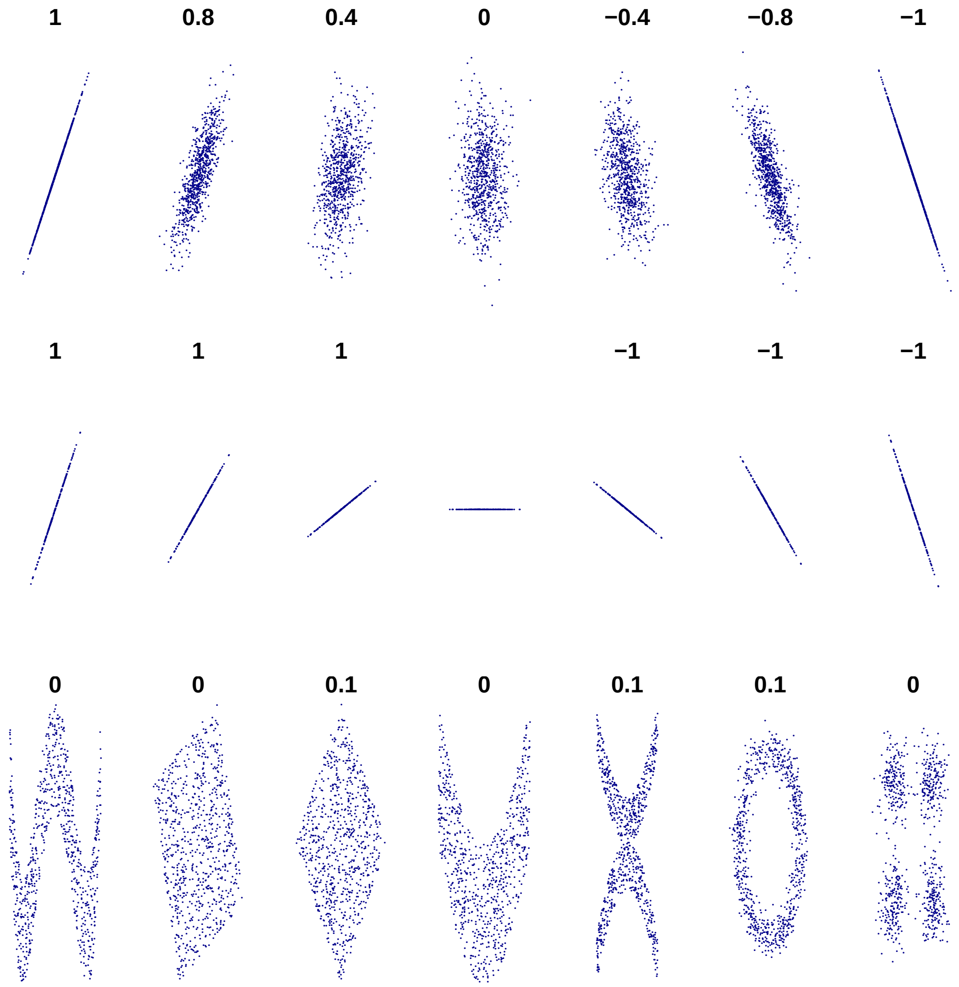


Figure 10.28: Examples of correlation between samples of two value pairs, plotted on x- and y-axis. Github–Local

Three commonly encountered correlation metrics are:

- *Pearson product-moment correlation coefficient*, (also known as Pearson's R or Pearson's r), which applies to continuous variables,
- Spearman's rho, ρ (a lowercase Greek letter), is identical to Pearson's coefficient except the correlation is calculated from the ranked values, i.e., the sorted order (which makes it immune to extreme values),
- Kendall's tau, τ (a lowercase Greek letter), is like Spearman's rho in that it is based on ranked values, but the calculation is based on the number of items sharing the same rank (i.e., relative difference in rank is not included in the calculation; Spearman's rho does include relative differences).

The `cor.test` function, included in the base system, supports all three coefficients and provides confidence interval.

Dichotomous variables: When the result of a measurement has one of two values, the standard techniques for calculating correlation, which require that most if not all values be unique, cannot be used. It is possible to recast the problem in terms of probabilities, which means that the approach taken for every problem could be different.

The following is an example of one approach to a particular binary problem involving binary measurements.

One technique for having high reliability access files, is to host the files on two or more websites; if one site cannot be accessed, the file could be obtained from another site. The naive analysis suggests, that, if the average reliability of the websites is 95%, then the reliability of two paired sites would be 99.75%. However, this assumes the unavailability of each website is independent of its paired site.

A study by Bakkaloglu, Wylie, Wang and Ganger[119] had a client program read a file from over 120 websites every 10-minutes, between September 2001 and April 2002. They recorded whether the file was successfully accessed or not.

Most websites were available most of the time. Bakkaloglu et al proposed various techniques for calculating correlated failures, based on the probability that site X is unavailable when site Y is unavailable, i.e., $P(X unavailable|Y unavailable)$. The following example takes the mean value over all pairs of sites:

$$= mean(P(X unavailable|Y unavailable))$$

$$= mean\left(\frac{P(X\&Y unavailable)}{P(Y unavailable)}\right)$$

The following calculates the average unavailability probability for one site paired with every other site; see Github–probability/reliability/web-avail.R:

```
given=web_down[ , ind]
others=web_down[ , -ind]

both_down=(others & given)

av_prob=mean(colSums(both_down)/sum(given))
```

Averaged over all pairs of sites the probability of one site being unavailable, when its pair is also unavailable, is 0.3 (at the 10-minute measurement point). Given that all accesses originated from the same client, it is not surprising that this probability is much higher than the average probability of one site being unavailable (0.1); all accesses start off going through the same internet infrastructure and problems in this infrastructure will affect access to all sites.

10.5.5 Contingency tables

Count data with categorical explanatory variables has a natural visual representation, as a table of numbers; these tables are known as *contingency tables*. Table 10.7 shows a count of items in the sample having both the listed row and column attributes.

Contingency tables are a technique for reducing lots of data to a compressed visual form. Reasons for compressing data to this form include: wanting to hide information (i.e., readers have to think about what is being presented), not knowing how to make the best use of available information (i.e., the compressed form throws away potentially useful information). Analysis of the uncompressed data is likely to reveal more about it, than an analysis of the simplified form.

Sometimes the only available data is present in a contingency table.

A study by Nightingale, Douceur and Orgovan[1361] investigated the characteristics of hardware failures over a very large number of consumer PCs. Table 10.7 shows a contingency table containing the available data, i.e., the number of system crashes believed to have been caused by hardware problems involving the system DRAM or CPU.

	DRAM failure	**no DRAM failure**
CPU failure	5	2,091
no CPU failure	250	971,191

Table 10.7: Number of system crashes of consumer PCs traced to CPU or DRAM failures. Data from Nightingale et al.[1361]

The traditional, manual friendly, technique for analyzing this kind of data is the chi-squared test (χ is the Greek letter), which provides a yes/no answer.[xvii]

[xvii] The `chisq.test` function is part of the base system; if your readership demands a chi-squared test, the `chisq_test` function in the `coin` package can be used to bootstrap confidence intervals.

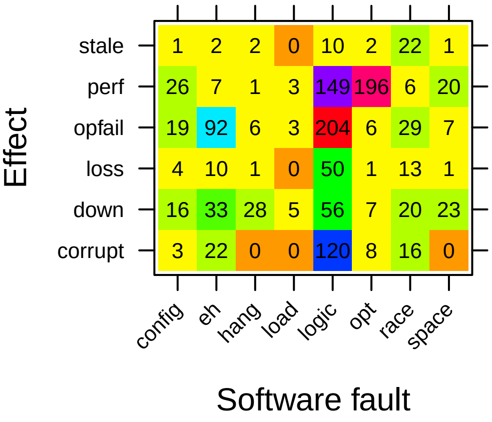
A regression model can be fitted to this data (even though there is not a lot of it), extracting more information than the chi-squared test. Github–Local

```
Call:
glm(formula = failures ~ CPU * DRAM, family = poisson, data = PC_crash)

Deviance Residuals:
[1]  0  0  0  0

Coefficients:
                  Estimate Std. Error   z value Pr(>|z|)
(Intercept)      13.786278   0.001015 13586.243  < 2e-16 ***
CPUTRUE          -6.140881   0.021892  -280.505  < 2e-16 ***
DRAMTRUE         -8.264818   0.063254  -130.661  < 2e-16 ***
CPUTRUE:DRAMTRUE  2.228858   0.452194     4.929 8.27e-07 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance:  2.6646e+06  on 3  degrees of freedom
Residual deviance: -7.7825e-11  on 0  degrees of freedom
AIC: 43.948

Number of Fisher Scoring iterations: 3
```

The information in the fitted model that is not immediately obvious from the numbers in the table, is that the crash rate is higher when both the CPU and DRAM fail (although, in this case, an obvious conclusion).

The regression approach makes it trivial to handle more rows and columns, as well as non-straight line fits. As the number of values in the table increases, it becomes more difficult to visually extract any patterns that may be present; figure 10.29 shows how a heatmap might be one way of highlighting details.

There are a wide variety of techniques for comparing multiple contingency tables. Note: different pair comparison algorithms can give very different results.[1792]



	config	eh	hang	load	logic	opt	race	space
stale	1	2	2	0	10	2	22	1
perf	26	7	1	3	149	196	6	20
opfail	19	92	6	3	204	6	29	7
loss	4	10	1	0	50	1	13	1
down	16	33	28	5	56	7	20	23
corrupt	3	22	0	0	120	8	16	0

Figure 10.29: Number of software faults having a given consequence, based on an analysis of faults in Cassandra. Data from Gunawi et al.[747] Github–Local

10.5.6 ANOVA

Readers are likely to encounter the acronym ANOVA (*Analysis of variance*), an analysis technique that developed independently of linear regression and having its own specialized terminology. This technique was designed for manual implementation.

Functionally ANOVA and least squares are both special cases of the general linear model (ANOVA is a special case of multiple linear regression with orthogonal, categorical predictors; ANCOVA adds covariates to mix). A one-way analysis of variance can be thought of as a regression model having a single categorical predictor, that has at least two (usually more) categories.

Treating the various kinds of ANOVA models as special cases of the family of regression models, makes it possible to use the more flexible options available in regression modeling (e.g., easier handling of unequal group sizes, adjusting for covariates and methods for checking models).

The anova function generates ANOVA style output, when passed a model built using `glm` and some other regression model building functions; the Anova function in the `car` package supports more functionality.

One-way ANOVA focuses on testing for differences among a group of means; it evaluates the hypothesis that $\alpha_i = 0$ in the following equation:

$Y_i = \mu + \alpha_i + \varepsilon_i$

where: μ is the group mean, α_i is the effect of the response variable on the i'th group and ε_i is the corresponding error.

Chapter 11

Regression modeling

11.1 Introduction

Regression modeling is the default hammer used in this book to create the output from data analysis of software engineering data; figure 11.1, gives a high level overview of the various kinds of hammers available in the regression modeling toolkit. Concentrating on a single, general technique, removes the need for developers to remember how to select from, and use, many special purpose techniques (which in many cases only return a subset of the information produced by regression modeling).

The arrow lines connect related regression techniques, based on the characteristics of the data they are designed to handle; the techniques highlighted in red are the common use cases for their respective data characteristics.

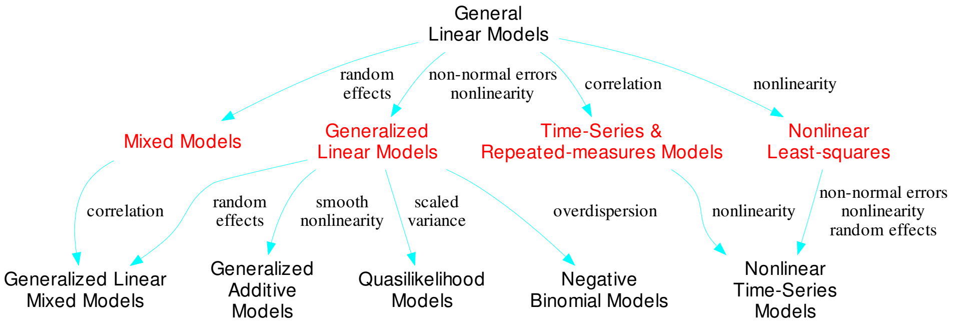

General Linear Models
random effects
non-normal errors nonlinearity
correlation
nonlinearity
Mixed Models
Generalized Linear Models
Time-Series & Repeated-measures Models
Nonlinear Least-squares
correlation
random effects
smooth nonlinearity
scaled variance
overdispersion
nonlinearity
non-normal errors nonlinearity random effects
Generalized Linear Mixed Models
Generalized Additive Models
Quasilikelihood Models
Negative Binomial Models
Nonlinear Time-Series Models

Figure 11.1: Relationship between data characteristics (edge labels) and applicable techniques (node labels) for building regression models.

Regression modeling is powerful enough to fit almost any data to within any selected error bounds, which means overfitting is an ever present danger;[i] model validation (e.g., how well a model might fit new data, or an estimation of the benefit obtained from including each coefficient in a model) is an important self-correcting step.

As always, it is necessary to remember the adage: "All models are wrong, but some are useful."

The main reasons for building a regression model are:

- understanding: structuring the explanatory variable(s) in an equation that can be used to interpret the impact they have on the response variable, i.e., build an understanding of the processes that influence the response variable to behaves the way it does,
- prediction: that is predicting the value of the response variable, for values of the explanatory variables that have not been measured.

The focus of interpretive modeling is understanding why, which creates a willingness to trade-off prediction accuracy for model simplicity, while the focus of predictive modeling is accuracy of prediction, which creates a willingness to trade-off understanding of behavior for greater accuracy.

[i] It is possible to fit an expression containing a single parameter to any data, to any desired degree of accuracy.[225]

Understanding is the primary focus for the model building in this book; builders of computing systems are generally interested in controlling what is happening and control requires understanding; predicting is a fall back position. Model building for prediction is often easier than building for understanding, once readers master building for understanding they will not find it difficult to switch to a predictive focus.

Regression models contain a *response variable*, one or more *explanatory variables*[ii], and some form of error term.

The *response variable* is modeled as some combination of *explanatory variables* and an additive or multiplicative error term (the error term associated with each explanatory variable represents behavior not accounted for by the explanatory variable; different kinds of regression model make different assumptions about the characteristics of the error).

It is always possible to concoct a model that fits some data to within any error tolerance, i.e., the amount of variation in the measurements used, that the model does not explain. It is very important to always ask how well a model is likely to fit all the data likely to be encountered, not just the data used to build it.

If a sample contains many variables, then it is sometimes possible to build a model only using a few of these variables, that has an impressive fit to the chosen response variable. A study by Zeller, Zimmermann and Bird[1985] built a fault prediction model whose performance was comparable to the best available at the time. The model used four explanatory variables to predict the probability of a fault report being associated with the source code contained a file; the explanatory variables were the percentage occurrence of each of the characters `IROP` in each file. The model was *discovered* by checking how good a job every possible character did at predicting fault probability, and picking those that gave the best fit.

11.2 Linear regression

The simplest form of regression model is linear regression, where the *response variable* is modeled as a linear combination of *explanatory variables* and an additive error (the error terms are assumed to be independent and identically distributed; ε denotes the total error). The equation is:

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + \varepsilon \tag{11.1}$$

Note that the term *linear* refers to the coefficients of the model, i.e., β, not the form taken by the explanatory variables, which may have a non-linear form, as in:

$y = \alpha + \beta x^2 + \varepsilon$

or:

$y = \alpha + \beta \log(x) + \varepsilon$

A linear model is perhaps the most commonly used regression model, reasons for this include:

- many real world problems exhibit linear behavior, or a good enough approximation to it for practical purposes, over their input range,
- they are much easier to fit manually than more sophisticated models, and until recently software to build other kinds of models was not widely available,
- they can generally be built with minimal input from the user (apart from having to decide which column of data to use as the response variable).

The `glm` function[iii] builds a linear model and the use common case requires two argument value, a formula expressing a relationship between variables (response variable on the left

[ii] Books that focus on the predictive aspect of models, use the term *prediction variable* or just *predictor*, while those that focus on running experiments use terms such as *control variables* or just the *controls*.

[iii] Many books start by discussing the `lm` function, rather than `glm`, because the mathematics that underpins it is easier to learn, another reason for this is herd mentality, it's what everybody else does; if you dear reader want to learn this mathematics I recommend taking this approach. As its name implies the Generalised Linear Method has a wider range of applicability and its use here is in line with the aim of teaching one technique that can be used everywhere. Also, the mathematics behind `glm` makes fewer assumptions about the sample characteristics, e.g., it does not have the precondition that the variance in the error to be constant (which `lm` does).

and explanatory variable(s) on the right), and an object containing the data (this object is required to contain columns whose names match the identifiers appearing in the formula). The formula has the form of an equation, with the = symbol replaced by ~ (pronounced *is distributed according to*) and the coefficients α and β are implicitly present, i.e., they do not need to be explicitly specified in the code.

The following code uses glm to build a model showing the relationship between the number of lines of source code (*sloc*) in FreeBSD, and the number of days elapsed since the project started (in 1993):

```
BSD_mod=glm(sloc ~ Number_days, data=bsd_info)
```

The fitted equation is:

$E[sloc] = \alpha + \beta \times Number\_days$

where: $E[sloc]$ is the expected value of *sloc* (the error term is discussed below).

Figure 11.2 shows the measured data points and a straight line based in the coefficients contained in the object returned by glm.

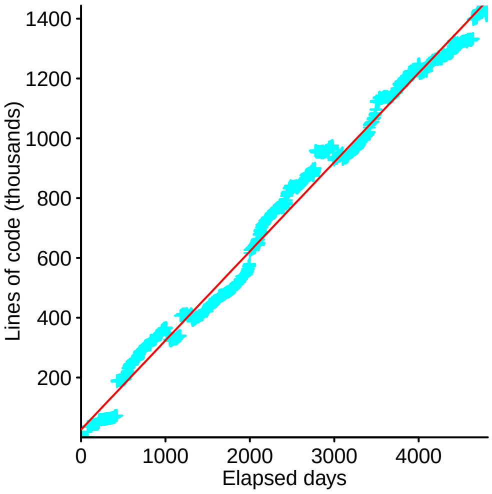


Figure 11.2: Total lines of source code in FreeBSD by days elapsed since the project started (in 1993). Data from Herraiz.[1785] Github–Local

The summary function takes the object returned by glm and prints details about the fitted model;[iv] the following is for the model fitted to the FreeBSD data: Github–Local

```
Call:
glm(formula = sloc ~ Number_days, data = kind_bsd)

Deviance Residuals:
   Min       1Q   Median       3Q      Max
-82990   -32136    -3609    35389    87324

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.139e+05  1.171e+03   97.24   <2e-16 ***
Number_days 3.937e+02  4.205e-01  936.33   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 1657283104)

    Null deviance: 1.4610e+15  on 4826  degrees of freedom
Residual deviance: 7.9964e+12  on 4825  degrees of freedom
AIC: 116172

Number of Fisher Scoring iterations: 2
```

The table following the Coefficients: header, in the summary output, lists the fitted values for α and β (Intercept and Number_days respectively), the standard error in these estimates (Std.Error) and the probability that, if the true value of the coefficient was zero, the estimated value would have occurred by chance (in the Pr(>|t|) column).

The values listed in the summary output can be plugged into the model formula to give the following fitted equation:

$$sloc = 1.139 \cdot 10^5 + 3.937 \cdot 10^2 Number\_days \tag{11.2}$$

The fit between the model and the data is not perfect, and the following are the two forms of uncertainty, or variation, present in the model:

1. Uncertainty in the values of the model coefficients. The values listed in the Std. Error column denote one standard deviation, which when added to the model gives the following:

$$sloc = (1.139 \cdot 10^5 \pm 1.171 \cdot 10^3) + (3.937 \cdot 10^2 \pm 4.205 \cdot 10^{-1}) Number\_days \tag{11.3}$$

2. Uncertainty caused by the inability of the explanatory variable used in the model to explain everything. This uncertainty is the ε appearing in equation 11.1; the term *residual* is used to denote this quantity. In the general case it is unlikely that ε will

[iv] Only a few digits of the estimated values are printed by default.

have a fixed value over the range of values supported by a model and glm does not return any value(s).

In figure 11.2 the variations in the unexplained error, ε, appear to be small. The aov function can be used to obtain a single fixed value; it returns 40,710 as the residual standard error. The equation, including this estimate of the residual is:

$sloc = 1.139 \cdot 10^5 + 3.937 \cdot 10^2 Number\_days \pm 4.071 \cdot 10^4$

In other words, the difference between measured values and values calculated using this fitted model are predicted to have a standard error of $4.071 \cdot 10^4$.

The object returned by the call to glm can be used to make predictions, and these can be overlaid on the output from an earlier call to plot, as follows:

```
BSD_pred=predict(BSD_mod)  # uses fitted model and measured values
lines(BSD_pred, col="red") # x-axis starts at 1 and increment
```

The predict/lines approach follows this book's aim of using techniques that work for the general case. Plotting a fitted straight line is such a common operation that there is a function for doing just that, e.g., abline(reg=BSD_mod, col="red"), but this does not always work when the axis have been scaled in some way and is of no use for fitted models that are more complicated than a straight line.

Before being carried away with the high degree of agreement between this model and the data, it is important to remember that the model has a number of characteristics that do not reflect reality, including:

- source code does not spontaneously grow of its own accord, and the only justification for treating *number of days* as an explanatory variable is that the resulting model provides potentially interesting insight into the rate of growth of these software systems.
- when it started the BSD project contained zero lines of code, but this model has an Intercept of $1.39 \cdot 10^5$,
- the model shows the number of lines increasing forever, at a constant rate, whereas at some point in the future growth must slow down and eventually stop,
- it says nothing about large amounts of code being added/removed over very short periods (known to exist because of visible breaks in the connectedness of plotted values).

While the model has various disconnects with reality, it does provide strong evidence that growth has been remarkable constant over a long period. Unless there are seismic changes within the FreeBSD development world, the constant rate of code growth would be expected to continue to hold for a non-trivial number of days into the future.

Fitting a model to the data marks the start of the next stage of analysis; creating viable explanations for the processes that could have produced the behavior found. Some factors and processes that might be involved in driving FreeBSD's essentially constant rate of growth over 20 years include:

- developers working on the system have continually discovered new functionality to add,
  - if there has always been functionality to add, why haven't more developers become involved, increasing the rate of growth until there is less to do?
  - to what extent is the continual stream of new hardware devices responsible for driving growth?
- what are the bottlenecks that have prevented increases in growth rate, when the resources have been available?
  - has growth rate remained constant because the developers working on the systems have remained constant?
  - is there a buffer of code waiting to be released, whose growing and shrinking hides an internal growth rate that is much more variable than the externally visible rate?

The questions answered by the analysis of one set of measurements invariable raises more questions, whose answers require more data.

The call to summary, passing the value returned by glm, is an example of function overloading in action. The value returned by glm has class glm, which, when passed as an argument to summary, results in summary.glm being called; a call to predict results in predict.glm being called (function overloading is the most common use of object-oriented constructs in R programs; the use of a period in the function name is a naming convention followed by the implementers and not something that changes the behavior of the R compiler).

Some readers of data analysis may find a visual presentation of the coefficients of a fitted model, along with their standard error, easier to process. The sjPlot package offers a variety of options for plotting of fitted model information.

11.2.1 Scattered measurement values

In the FreeBSD analysis, the measurements ran together in a way that created a visually recognizable line. The common case is not always so accommodating, and often when many samples are plotted a scattering of visually disjoint points appears; viewed as a whole a general trend may emerge.

A study by Kampstra and Verhoef[957] investigated the estimated cost and duration of 73 large Dutch federal IT projects.[v] Figure 11.3 shows that few measurement points are close to the (red) fitted line returned by glm; the variability of measured values is much larger than that for the FreeBSD data. While numeric estimates of the uncertainty present in the fitted model are readily available, interpreting these numeric values requires a degree of effort and some experience. A confidence interval provides an easy to interpret visual representation of the uncertainty in a fitted model.

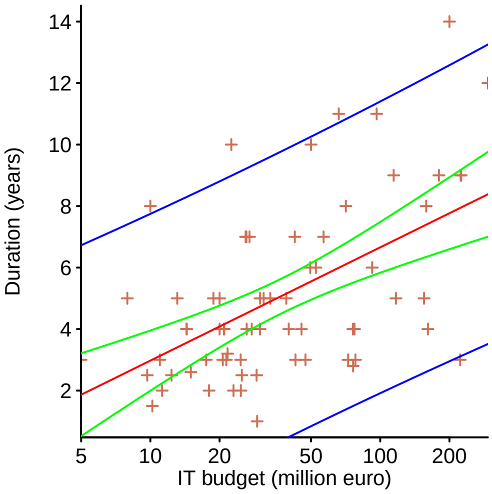


Figure 11.3: Estimated cost and duration of 73 large Dutch federal IT projects, along with fitted model and 95% confidence intervals (green for the bounds of the fitted line and blue for the bounds of any new measurements). Data from Kampstra et al.[957] Github–Local

The kind of uncertainty, in the fitted model, of interest will depend on whether the model is built to gain understanding or make predictions:

- when understanding is the priority, the confidence interval of interest involves the estimated model coefficients:
  - a call to predict with the argument se.fit=TRUE, returns the standard error for each fitted value. Multiplying se.fit by qnorm,[vi] converts the returned value to a 95% confidence interval (in this case, 2.5% above and below the fit; the two qnorm values differ only in sign because the Normal distribution is symmetrical), i.e., there is a 95% expectation that the actual model fits within the interval enclosed by these lower/upper bounds. qnorm(0.975)==1.96 and the literal value is often used (sometimes the value 2 is treated as a sufficiently close approximation).[vii]

    ```
    fed_pred=predict(fed_mod, newdata=list(log.IT=1:7, log.IT_sqr=(1:7)^2),
                                                     se.fit=TRUE)
    lines(fed_pred$fit, col="green")      # fitted line
    # CI above and below
    lines(fed_pred$fit+qnorm(0.975)*fed_pred$se.fit, col="green")
    lines(fed_pred$fit+qnorm(0.025)*fed_pred$se.fit, col="green")
    ```

  - the confint function in the MASS package, or the boot.ci function in the boot package, can be used to obtain a point estimate of the confidence interval of the fitted model coefficients.
- when prediction is the priority, the interval is known as the *prediction interval*; the bounds between which newly measured values are expected to appear. Two sources of uncertainty are added to calculate the prediction interval: uncertainty in the model coefficients (i.e., the confidence interval) plus the variance in the data not explained by the fitted model; the calculation is (the predict function can perform this calculation for a few types of fitted models):

  ```
  # print.aov also calculates it from residuals returned by glm...
  MSE=sum(fed_mod$residuals^2)/(length(fed_mod$residuals)-2)
  # Variances, but not sd, can be added
  pred_se=sqrt(fed_pred$se.fit^2+MSE)
  lines(fed_pred$fit+1.96*pred_se, col="blue")
  lines(fed_pred$fit-1.96*pred_se, col="blue")
  ```

When measurement values, and an associated fitted regression line, are plotted, it is easy to visually fixate on the line and forget about the associated uncertainties. Including a confidence band as part of a plot provides a vivid visual reminder of the uncertainty in the fit.

[v] They discovered there was a lot of uncertainty in the estimates given.

[vi] This calculation assumes that the measurement error has a Normal distribution, the default assumption made by glm when building a model.

[vii] For small sample sizes a call to qt may be more accurate.

Plotting values does not always reveal an obvious pattern in the distribution of points. The absence of a visual pattern may be because no relationship exists between the response and explanatory variables, or because the noise in the data is much greater than the signal (i.e., a relationship that exists is swamped by noise present in the measurements).

How much random scattering of measurement values has to exist before a fitted regression model can be said to be not worth bothering about?

The glm function, and many other model building functions available in R, is capable of fitting models to data points that are randomly distributed. For instance, Figure 11.4 shows the number of updates and fixes made in various Linux versions released between early 2011 and 2012. The standard error of the fitted line shows that its slope could have a positive or negative value.

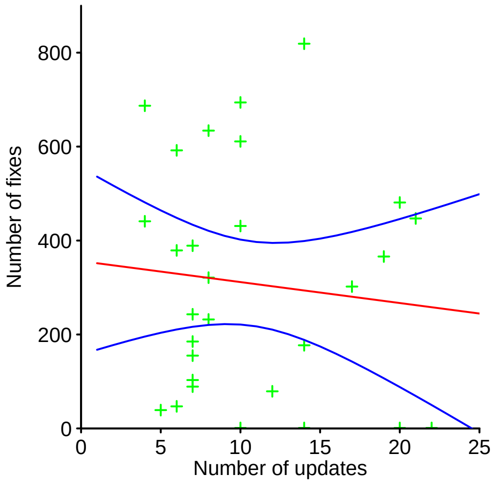


Figure 11.4: Number of updates and fixes in each Linux release between version 2.6.11 and 3.2. Data from Corbet et al.[395] Github–Local

The output from summary shows how poor the fit actually is;the Pr(>|t|) column lists the p-value for the hypothesis that the coefficient in the corresponding row is zero, i.e., that no relationship was found to exist for that component of the model. Github–Local

```
Call:
glm(formula = Fixes ~ Total.Updates, data = cleaned)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-310.60  -223.67     0.48   184.51   525.26

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)    356.233    101.522   3.509   0.0016 **
Total.Updates   -4.464      8.478  -0.526   0.6029
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 60685.71)

    Null deviance: 1655335  on 28  degrees of freedom
Residual deviance: 1638514  on 27  degrees of freedom
AIC: 405.62

Number of Fisher Scoring iterations: 2
```

11.2.2 Discrete measurement values

Regression models are not limited to fitting continuous numeric explanatory variables, variables having nominal values (i.e., discrete) can also be included in a fitted model.

A study by Cook and Zilles[389] investigated the impact of compiler optimization flags on the ability of software to continue to operate correctly, when subject to random bit-flips, i.e., simulating random hardware errors; 100 evenly distributed points in the program were chosen and 100 instructions from each of those points were used as fault injection points, giving a total of 10,000 individual tests run, for each of 12 programs from the SPEC2000 integer benchmark compiled using gcc version 4.0.2 (using optimization options: 00, 02 and 03) and the DEC C compiler (called *osf*).

The fitted model has percentage of correct benchmark program execution as the response variable[viii] and optimization level as the explanatory variable; the call to glm is unchanged:

```
bitflip_mod=glm(pass.masked ~ opt_level, data=bitflip)
```

The summary output of the fitted model is: Github–Local

```
Call:
glm(formula = pass.masked ~ opt_level, data = bitflip)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-12.6689   -2.8454   -0.3478    4.4017    8.1100
```

[viii] Percentage correct is always between 0 and 100%; technically correct techniques for handling response variables having a lower and upper bound are discussed in section 11.3.6.

```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)    28.589      1.825  15.665  < 2e-16 ***
opt_level02     9.161      2.581   3.550  0.00112 **
opt_level03     7.429      2.581   2.878  0.00677 **
opt_levelosf   11.642      2.414   4.822 2.74e-05 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 29.97578)

    Null deviance: 1785.8  on 38  degrees of freedom
Residual deviance: 1049.2  on 35  degrees of freedom
AIC: 249.07

Number of Fisher Scoring iterations: 2
```

Plugging the model coefficients into the regression equation we get:

$pass.masked = 28.6 + 9.2 \times D_{O2} + 7.4 \times D_{O3} + 11.6 \times D_{osf}$

where: D_i, known as a *dummy variable* or *indicator variable*, take one of two values:

$$D_i = \begin{cases} 1 & \text{optimization flag used} \\ 0 & \text{optimization flag not used} \end{cases}$$

The value for optimization 00 is implicit in the equation, it occurs when all other optimizations are not specified, i.e., its value is that of the intercept.

The standard error in the 02 and 03 compiler options is sufficiently large for their respective confidence bounds to have significant overlap; suggesting that these two options have a similar impact on the behavior of the response variable.

11.2.3 Uncertainty only exists in the response variable

Many algorithms used to fit regression models attempt to minimise the difference between the measured points and a specified equation. For instance, least-squares minimises the sum of squares of the distance along one axis between each data point and the fitted equation;[ix] alternative minimization criteria are discussed later, e.g., giving greater weight to positive error than negative error.

An important, and often overlooked, detail, is that many regression techniques assume that the values of the explanatory variable(s) contain no uncertainty (i.e., measurements are exact), with all uncertainty, ε, occurring in the response variable (see equation 11.2).

A consequence of assuming uncertainty only exists in the response variable, is that the equation produced by fitting a model that specifies, say, X as the explanatory variable and Y the response variable will not be algebraicly consistent with a model that assumes Y is the explanatory variable and X the response variable. That is, algebraicly transforming the first equation produces an equation whose coefficients are different from the second.

A study by Kroah-Hartman[1033] investigated the number of commits made between the release of a version of Linux and the immediately previous version, and the number of developers who contributed code to that release, for the 67 major kernel releases between versions 2.6.0 and 4.6.

In the upper plot of figure 11.5, the number of developers is treated as the explanatory variable (x-axis), and number of commits as the response variable (y-axis), with the fitted regression line in red and dashed lines showing the difference between measurement and fitted model. In the lower plot the explanatory/response roles played by the two variables, when fitting the regression model, is switched; to simplify comparison the axis denote the same variables in both plots, with the blue line denoting the newly fitted model, and dashed lines showing the difference between measurement and model (now on the x-axis response variable; the line fitted in the upper plot is also plotted for comparison, still in red).

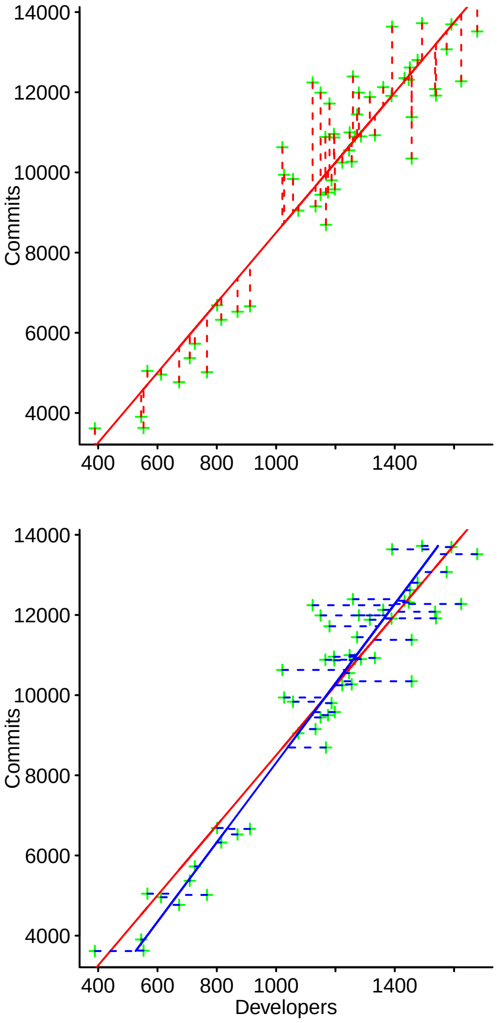


Figure 11.5: Number of commits made, and the number of contributing developers for Linux versions 2.6.0 to 3.12. The blue line in the right plot is the regression model fitted by switching the x/y values. Data from Kroah-Hartman.[1033] Github–Local

[ix] Minimising the sum of squares in the error has historically been popular because it is a case that can be analysed analytically.

In the first case the fitted equation is:

$$commits = -237 \pm 523 + (8.7 \pm 0.44) mathitNumber\_devs \tag{11.4}$$

transforming this equation we get:

$$\begin{aligned} Number\_devs &= \frac{237 + commits}{8.7} \\ &= 27 + 0.11 commits \end{aligned} \tag{11.5}$$

However, when a model is fitted by switching the roles of the two variables, in the formula passed to `glm`, the model returned is described by the following equation:

$Number\_devs = 162 \pm 52 + (0.10 \pm 0.005) commits$

which differs from equation 11.5, obtained by transforming equation 11.4.

There is another difference between the two fitted models, the second model is a better fit to the data. Somebody who is only interested in the quality of fit may be tempted to select the second model, purely for this reason.

What is the procedure for deciding which measurement variables play the role of response and explanatory variable, e.g., should number of developers be considered an explanatory or response variable?

An important attribute of explanatory variable(s) is that their value is controlled by the person making the measurement. For instance, the model building process used to create figure 11.2 has number of days as the explanatory variable; this choice was completely controlled by the person making the measurements.

The Kroah-Hartman commit measurements are based on the day of release of a version of the Linux kernel, a date that is outside the control of the measurement process. In fact both measurements have the characteristics of a response variables, that is, the value they have, was not selected by the person making the measurement. Both the possibility of variation in Linux version release dates and variation in number of commits made by developers are sources of uncertainty, both variables need to be treated as containing measurement error.

Building a regression model using explanatory variables containing measurement error can result in models containing biased and inconsistent values, as well as inflating the Type I error rate.[263, 1662]

There are a variety of regression modeling techniques that can take into account error in the explanatory variable. These techniques are sometimes known as *model II* linear regression techniques (model I being the case where there is no uncertainty in the explanatory variables), and also as *errors-in-variable models*, *total least-squares* or *latent variable models*; methods used go by names such as *major axis*, *standard major axis* and *ranged major axis*.

If all the variables used to build a model contain some amount of error, then it is necessary to decide how much error each variable contributes to the total error in the model. Some model II techniques are not scale invariant, that is, they are only applicable if both axes are dimensionless or denote the same units, otherwise rescaling one axis (e.g., converting from kilometers to miles) will change its relative contribution. If each axis denotes a different unit, it does not make sense to use a model building technique that attempts to minimise some measure of combined uncertainty.

SIMEX (SIMulation-EXtrapolation) is a technique for handling uncertainty in explanatory variables that works in conjunction with a range of regression modeling techniques. The SIMEX approach does not suffer from many of the theoretical problems that other techniques suffer from, but requires that the model builder provide an estimate of the likely error in the explanatory variable(s). The `simex` package implements this functionality, and supports a wide variety of regression models built by functions from various packages.

Continuing with the Linux developer/commit count example, to build a regression model using SIMEX, we need an estimate of the uncertainty in the number of developers contributing at least one commit to any given release. The `simex` function taking a model built using `glm` (and by other regression model building functions) and an estimate of the uncertainty in one or more of the explanatory variables, and returns an updated model that has been adjusted to take this uncertainty into account.

The following is a rough and ready approach to estimating the uncertainty in the Kernel attributes, measured by Kroah-Hartman:

- the release date of a new version of Linux is assumed to have an uncertainty of ±14 days about the actual release date.[x]
- the possible variation in the unique contributor count for any release is assumed to be uniformly distributed in the range: measured contributor count plus/minus number of developers contributing their first commit in the last 14 days.
- based on these assumptions, a standard deviation of 41 is obtained for the number of unique developers making at least one commit, averaged over all versions (see Github–regression/clean/dev-commit.R).

Integrating this estimate of the standard deviation in the explanatory variable into a regression model is a two-step process:

- first build a regression model using `glm` in the usual way, but with the optional named parameter `x` set to `TRUE` (`y` also needs to be `TRUE`, but this is its default value and so the assignment below is redundant),
- pass the model returned by `glm` to `simex`, along with the name of the explanatory variable and its estimated standard deviation.

In code, the implementation is:

```
yx_line = glm(commits ~ developers, x=TRUE, y=TRUE)

sim_mod=simex(yx_line, SIMEXvariable="developers", measurement.error=41)
```

Compare equation 11.4 with the following equation, derived from the model returned by `simex` (see Github–regression/dc-simex.R):[xi]

$$commits = -387 \pm 453 + (8.9 \pm 0.4) Number\_devs$$

The error in individual explanatory variable measurements can be specified by assigning a vector to `measurement.error` (the argument `asymptotic=FALSE` is also required); see fig 11.35.

How reliable is a fitted model that ignores any uncertainty/error in explanatory variable measurements? The only way to answer this question is to build a model that takes this error into account and compare it with one that does not. The difference between the two ways of structuring fitted models can sometimes be much larger than that in figure 11.5.

The question of whether economies of scale exist for software development, can be answered by analysing project effort/size data. Figure 11.6 shows lines for two fitted regression models, one with Effort as the explanatory variable, the other with Size as the explanatory variable (from a study by Jørgensen, Indahl and Sjøberg[941]). If economies of scale exist, the slope of the effort/size line will be less than one (diseconomies of scale produce a slope greater than one). In this case, one slope is less than one and the other greater than one. The models fitted by switching response/explanatory variables are outside each other's 95% confidence intervals (there is no reason to expect them to be inside).

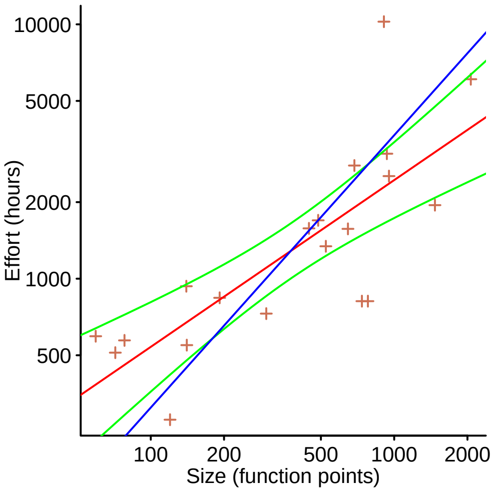


Figure 11.6: Effort/Size of various projects and regression lines fitted using Effort as the response variable (red, with green 95% confidence intervals) and Size as the response variable (blue). Data from Jørgensen et al.[941] Github–Local

Many measurement values treated as explanatory variables in this book were not under the control of the person who measured them. For instance, lines of code, number of files and reported problems measured at a given point in time are all response variables. To reduce your author's workload, most model fitting in this book does not make any adjustments for errors in the explanatory variables.

There are techniques, and `R` packages, for building complete models starting from the data, rather than refitting an existing regression model. For those wanting to a build model from scratch, the `lmodel2` package provides functions that implement many of the available methods.

[x] Pointers to a more reliable, empirically derived, value are welcome.

[xi] Readers might like to experiment with the value of `measurement.error`, to see the impact on the model coefficients.

11.2.4 Modeling data that curves

A model based on a straight line is a wonderful thing to behold, it is simple to explain and often aligns with people's expectations (many real world problems are well fitted by a straight line). However, life is complicated and throws curved data at us.

Having encountered an operating system having constant lines of code growth over many years, it is tempting to draw a conclusion about the growth rate of other operating systems. However, the way in which the data points curve around the fitted line in the upper plot of figure 11.7, suggests that some of the processes driving the growth of the Linux kernel are different from those driving FreeBSD; perhaps a quadratic or exponential equation would be a better fit (these possibilities were chosen because they are two commonly occurring forms for upwardly curving data).

This section is about fitting linear models, so the possibility of an exponential fit is put to one side for the time being; building non-linear models, including better fitting non-linear models to this data, is discussed later (see section 11.5).

The following call to `glm` fits an equation that is quadratic in the variable `Number_days`; the righthand side of the formula contains `Number_days+I(Number_daysˆ2)`. The `I` (sometimes known as *as-is*) causes its argument to remain unevaluated and is treated as a distinct explanatory variable (the ˆ operator has a distinct meaning within `glm`'s formula notation, see table 11.2, and use of `I` prevents the expected binary operator usage being overridden). An alternative way of including a squared explanatory variable in the model, is to assign the value `Number_daysˆ2` to a new variable and include this new variable's name on the righthand side of the formula. Figure 11.7, lower plot, shows the result of fitting this equation.

```
linux_mod=glm(sloc ~ Number_days+I(Number_days^2), data=linux_info)
```

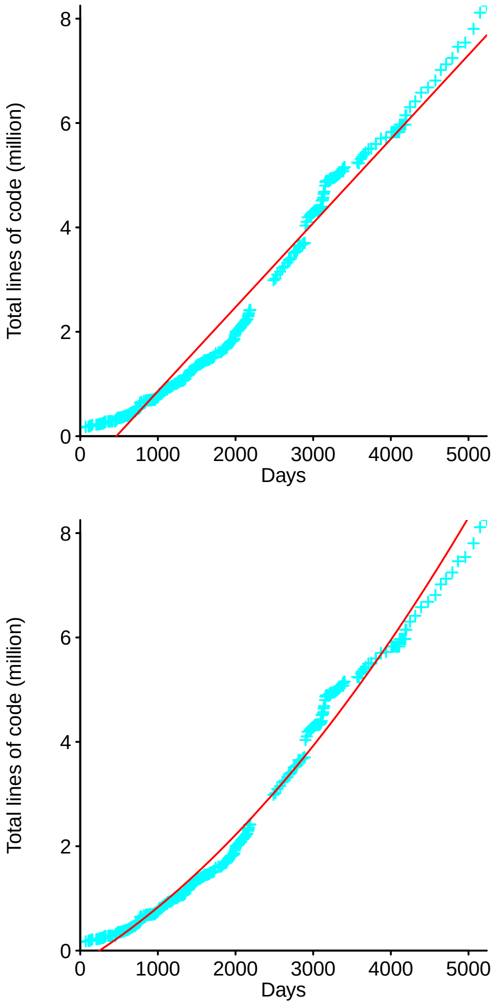


Figure 11.7: Lines of code in every initial release (i.e., excluding bug-fix versions of a release) of the Linux kernel since version 1.0, along with fitted straight line (upper) and quadratic (lower) regression models. Data from Israeli et al.[891] Github–Local

The quadratic fit looks like it is better than linear, but perhaps a cubic, quartic or higher degree polynomial would be even better fits. The higher the order of the polynomial used, the smaller the error between the fitted model and the data. The error decreases because the additional terms make it possible to do a better job of following the random fluctuations in the data. A method of applying Occam's razor is needed, to select the number of terms that produces the simplest model consistent with the data, and having an acceptable error.

The Akaiki Information Criterion, AIC, is a commonly used metric for comparing two or more models (available in the `AIC` function). It takes into account both how well a model fits the data, and the number of free coefficients in the model (i.e., constants selected by the model building process, such as polynomial coefficients); free coefficients have to pay their way by providing an appropriate improvement in a model's fit to the data.[xii] AIC can also be viewed as the information loss when the true model is not among those being considered.[275]

One set of selection criteria[275] are that models whose AIC differs by less than 2 are more or less equivalent, those that differ by between 4 and 7 are clearly distinguishable, while those differing by more than 10 are definitely different.

How much better does a quadratic equation fit Linux SLOC growth, compared to a straight line and how much better do higher degree polynomials fit? The following list gives the AIC for models fitted using polynomials of degree 1 to 4 (lower values of AIC are better). After initially decreasing the AIC starts to increase, once a fourth degree polynomial is reached; the third degree polynomial is thus the chosen linear polynomial, of those tested (other forms of equation could provide better fits using fewer free coefficients.) Github–Local

```
[1] Degree 1, AIC= 13998.0004739753
[1] Degree 2, AIC= 13674.6883243397
[1] Degree 3, AIC= 13220.8542892188
[1] Degree 4, AIC= 13221.7072389496
```

The following is the `summary` output for the fitted cubic model: Github–Local

```
Call:
glm(formula = LOC ~ Number_days + I(Number_days^2) + I(Number_days^3),
```

[xii] A negative AIC value may be the result of a nominal explanatory variable having many values, e.g., dates that are represented as strings, which are could be converted using `as.Date`.

```
    data = latest_version)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-428217   -80061     6503    64889   620500

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)
(Intercept)       3.432e+05  2.876e+04  11.935  < 2e-16 ***
Number_days      -3.664e+02  5.144e+01  -7.123 3.79e-12 ***
I(Number_days^2)  8.167e-01  2.456e-02  33.258  < 2e-16 ***
I(Number_days^3) -9.184e-05  3.371e-06 -27.242  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 23001633959)

    Null deviance: 1.8867e+15  on 494  degrees of freedom
Residual deviance: 1.1294e+13  on 491  degrees of freedom
AIC: 13221

Number of Fisher Scoring iterations: 2
```

Regression modeling produces the best fit for an equation over the range of data values used, and AIC helps prevent overfitting. No claims are made about how well the model is likely to fit data outside the range values used to fit it. Using a model, optimized to fit the available data, to make predictions outside the interval of the data values used can produce unexpected results.

What is the behavior of the above cubic model outside its fitted range, and in particular, what predictions does it make about future growth? Figure 11.8 shows that the model predicts a future decrease in the number of lines of code. A decreasing number of lines is the opposite of previous behavior, this prediction does not appear believable (if this decreasing behavior were predicted by a more detailed model of code growth, that closely mimicked real-world development by using information on the number of developers actively involved, and a list of functionality likely to be implemented, it would be more believable).

A quadratic equation might not fit the data as well as a cubic equation, but the form of its predictions (increasing growth) is consistent with expectations.

If the purpose of modeling is to gain understanding, then the quadratic model maps more closely to anticipated behavior; if the purpose is prediction within the interval of the fitted data, then the cubic model is likely to have a smaller error.

What about fitting other kinds of equations to the data? Equations such as $Y = \alpha e^{\beta X} + \varepsilon$ and $Y = \alpha X^{\beta} + \varepsilon$ are nonlinear in β; non-linear model building is discussed in section 11.5.

For a software system to grow, more code has to be added to it than is deleted. A constant rate of growth suggests either a constant amount of developer effort, or a bottleneck holding things up; an increasing rate of growth (i.e., quadratic) suggests an increasing rate of effort. The different code growth pattern seen in the Linux kernel, compared to NetBSD/FreeBSD and various other applications, has been tracked down[681] to device driver development; new hardware devices often share many interface similarities with existing devices; for Linux developers tend to copy an existing driver, modifying it to handle the hardware differences. It is this reuse of existing code that is the source of what appears to be a non-linear growth in developer effort. This method of creating a new device driver, performed by many developers working independently, can continue for as long as the new devices coming to market have common interfaces.

A linear regression model is not restricted to combining explanatory variables using polynomials, any function can be used as long as the coefficients of the model occur in linear form. For instance, the FreeBSD model plotted in figure 11.2 might include a seasonal term that varies with time of year; while a model containing the term $A\sin(2\pi ft + \phi)$[xiii] is nonlinear (because of ϕ, the phase shift), it can be written in linear form the follows:

$$A\sin(2\pi ft + \phi) = \alpha_s \sin(2\pi ft) + \alpha_c \cos(2\pi ft)$$

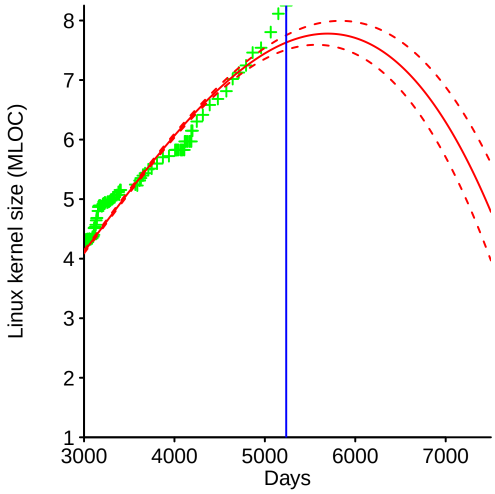


Figure 11.8: Actual (left of vertical line), and predicted (right of vertical line) total lines of code in Linux at a given number of days since the release of version 1.0, derived from a regression model built from fitting a cubic polynomial to the data (dashed lines are 95% confidence bounds). Data from Israeli et al.[891] Github–Local

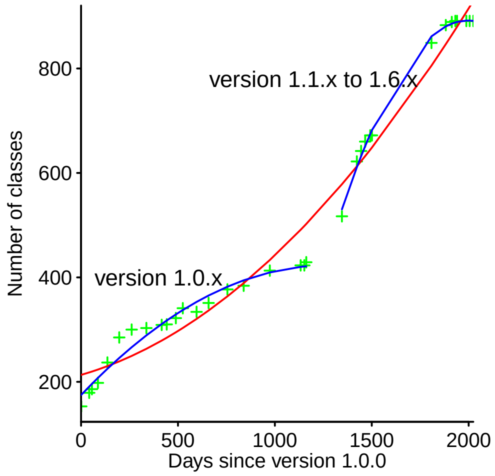


Figure 11.9: Number of classes in the Groovy compiler at each release, in days since version 1.0. Data From Vasa.[1864] Github–Local

[xiii] Some books use $A\cos(2\pi ft + \phi)$, which changes the phase by 90°and flips some signs.

where: $\alpha_s = A\cos\phi$ and $\alpha_c = A\sin\phi$; $A = \sqrt{\alpha_s^2 + \alpha_c^2}$ and $\phi = \arctan\frac{\alpha_s}{\alpha_c}$.

The call to glm is now (the argument to the trig functions is in radians):

```
rad_per_day=(2*pi)/365
freebsd$rad_Number_days=rad_per_day*freebsd$Number_days
season_mod=glm(sloc ~ Number_days+sin(rad_Number_days)+cos(rad_Number_days),
                      data=freebsd)
```

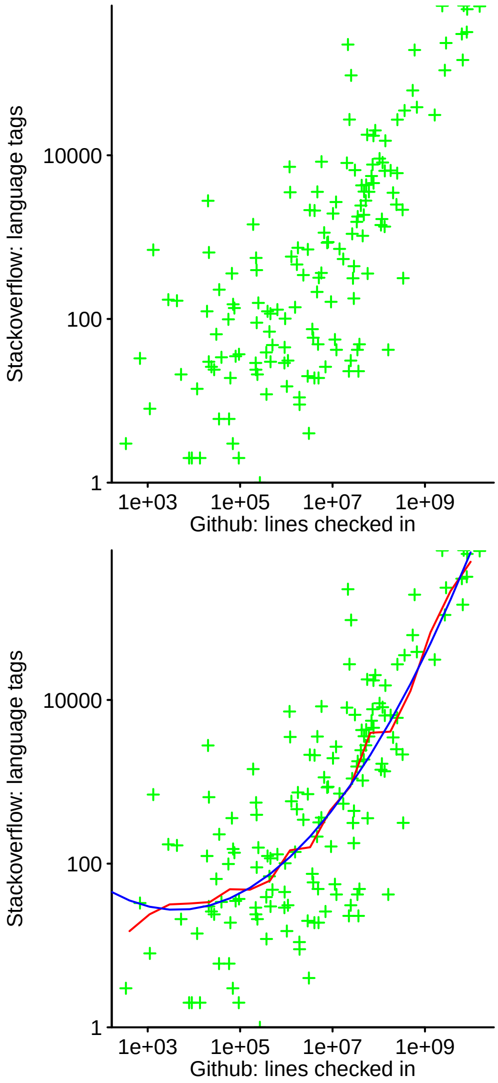


Figure 11.10: For each distinct language, the number of lines committed on Github, and the number of questions tagged with that language. Data from Kunst.[1046] Github–Local

The summary output, from the fitted model, shows that while a seasonal component exists, its overall contribution is small (see Github–regression/Herraiz-BSD-season.R).

While fitting a model using all available measurements points is a reasonable first step, subsequent analysis may suggest that the data might best be treated as two or more disjoint samples. There may be time dependent factors that have a strong influence on growth patterns.

Figure 11.9 shows the number of classes in the Groovy compiler at each release, in days since version 1.0. There are noticeable kinks in the growth rate at around 1,300 and 1,500 days. Fitting a model to the complete sample, shows upward trending quadratic growth in the number of classes over time, but fitting separate models to two halves of the sample, shows quadratic growth that flattens out.

An investigation the Groovy compiler developer, finds that the kink occurs at a transition between version numbers. It is possible to invent a variety of explanations for the pattern of behavior seen, but treating the measurements as-if they came from a single continuously developed code base, is probably not one of them; further investigation of the circumstances behind the development of the Groovy compiler is needed, to obtain the desired level of confidence in one of these, or other, models.

11.2.5 Visualizing the general trend

Even when the measurement points are scattered in what appears to be a general direction, it is little work to confirm this general trend.

A general technique for highlighting the trend followed by data is to fit a regression model to a consecutive sequence of small intervals of the data, joining this sequence of fits together to form a continuous line. Two methods based on this idea (both fitting such that the lines smoothly run together), are LOWESS (LOcally WEighted Scatterplot Smoothing) and LOESS (LOcal regrESSion); lowess and loess are the respective functions, with loess being used in this book.

A study by Kunst[1046] counted, for 148 languages, the number of lines committed to Github (between February 2013 and July 2014), and the number of questions tagged with that language name on Stackoverflow.

Figure 11.10, upper plot, shows lots of points that look as-if they trend along a straight line. The loess fit, red line in lower plot, shows the trend having a distinct curve. Experimenting with a quadratic equation in log(lines_committed) shows (blue line in lower plot) that this more closely follows the loess fit, than a straight line (a quadratic fit also has a lower AIC than a linear one; see Github–regression/langpop-corger-nl.R).

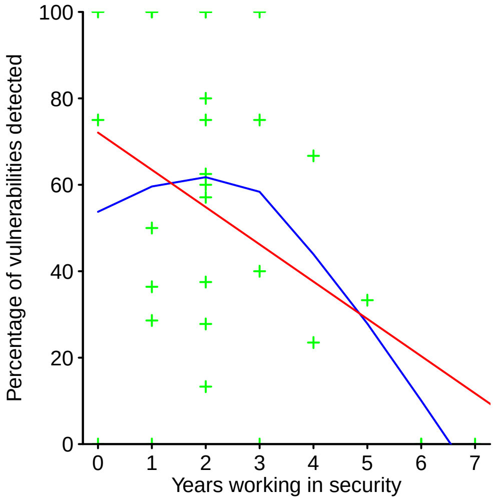


Figure 11.11: Percentage of vulnerabilities detected by developers who have worked a given number of years in security. Data extracted from Edmundson et al.[523] Github–Local

A call to loess has the same pattern as a call to glm, with the possible addition of an extra argument; span is used to control the degree of smoothing:

```
loess_mod=loess(log(stackoverflow) ~ log_github, data=langpop, span=0.3)
x_points=1:max(langpop$log_github)
loess_pred=predict(loess_mod, newdata=data.frame(log_github=x_points))
lines(exp(x_points), exp(loess_pred), col=pal_col[1])
```

A study by Edmundson, Holtkamp, Rivera, Finifter, Mettler and Wagner[523] investigated the effectiveness of web security code reviews, asking professional developers to locate vulnerabilities in code.

The lowess fit, blue line in figure 11.11, suggests that the percentage of vulnerabilities found increases as the number of years working in security increases, but then rapidly decreases. This performance profile seems unrealistic. A fitted straight line, in red, shows a decreasing percentage with years of work in the security field (its p-value is 0.02).

Perhaps the correct interpretation of this data, is that average performance does increase with years worked in the field, but that the subjects with many years working in security, who took part in the study, were more managerial and customer oriented people (who had time available to take part in the experiment), i.e., this data contains sampling bias. At the time of the study, software security work was rapidly expanding, so the experience profile is likely to be skewed with more subjects being less experienced.

When it is not necessary to transform either argument, the value returned by the `loess.smooth` function can be passed directly to `lines`.

```
lines(loess.smooth(dev$experience, dev$written, span=0.5), col=pal_col[2])
```

A `loess` visualization can also helpful when the number of data points is so large, they coalesce into formless blobs. The Ultimate Debian Database (see fig 11.23) is an example.

The `loess` function divides the range of x-axis values into fixed intervals, which means that when the range of x-values varies by orders of magnitude, the fitted curve can appear over stretched at the low values and compressed at high values.

One solution is to reduce the range of x-values by, for instance, taking the log, smoothing, and then expanding (see Github–regression/java-api-size.R); the following code is used in figure 11.32:

```
t=loess.smooth(log(API$Size), API$APIs, span=0.3)
lines(exp(t$x), t$y, col=loess_col)
```

11.2.6 Influential observations and Outliers

Influential observations are observations that have a disproportionate impact on the values of a model's fitted coefficients, e.g., a single observation significantly changes the slope of a fitted straight line. The terms *leverage* (or, based on the mathematical symbol used, *hat-value*) refer to the amount of influence a data point has on a fitted model; the `hatvalues` function takes the model returned by `glm` and returns the leverage of each point.

Influential observations might be removed or modified, or a regression technique used that reduces the weight given to what are otherwise overly influential points, e.g., the `glmrob` function in the `robustbase` package (which is not always as robust as desired and manual help may be required; see Github–regression/a174454-reg.R).

Outliers are discussed as a general issue in section 14, this subsection discusses outliers in the context of regression modeling. In this context an outlier might be defined as a data point having a disproportionately large standardized residual (here Studentized residuals are used). To repeat an important point made in that chapter: excluding any influential observations or outliers from the analysis is an important decision that needs to be documented in the results.

Cook's distance (also known as *Cook's D*) is a commonly used metric, which combines leverage and outlierness into a single number.

A study by Fenton, Neil, Marsh, Hearty, Radliński and Krause[585] involved data from 31 software systems for embedded consumer products. Figure 11.12 shows development effort against the number of lines of code, along with a fitted straight line and standard error bounds. At the right edge of the plot are two projects that consumed over 50,000 hours of effort, and the number of lines of code for these projects looks very small in comparison with other projects. Is the fitted model overly influenced by these two projects and should they be ignored or adjusted in some way?

As the number of points in a sample grows, there is an increasing probability that one or more of them will be some distance away from the fitted line; in any large sample a few apparent outliers are to be expected as a natural consequence of the distribution of the error. The following analysis illustrates the dangers of not taking sample size into account, when making judgements about the outlier status of a measurement point.

Figure 11.13 shows the result of building a model, after removing measurements having both a high Cook's distance and Studentized residuals, and repeating the process until points stop being removed. At the end of the process, most measurement points have been removed.

Removing overly influential points until everything looks respectable is seductive, it is an easy-to-follow process that does not require much thought about the story that the data might have to tell. For those who don't want to think about their data, the `outlierTest`

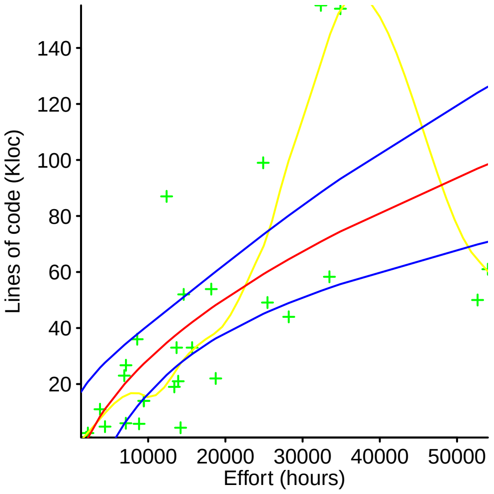


Figure 11.12: Hours to develop software for 29 embedded consumer products, and the amount of code they contain, with fitted regression model and loess fit (yellow). Data from Fenton el al.[585] Github–Local

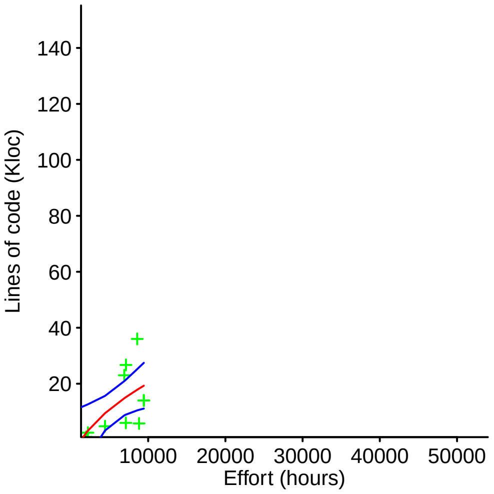


Figure 11.13: Points remaining after removal of overly influential observations, repeatedly applying Cook's distance and Studentized residuals. Data from Fenton el al.[585] Github–Local

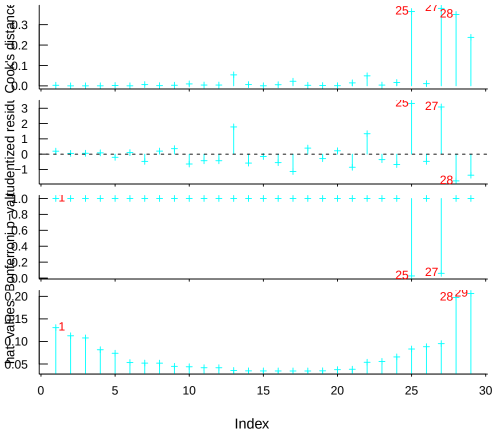


Figure 11.14: `influenceIndexPlot` for the model having the fitted line shown in figure 11.12; top three data points highlighted. Data from Fenton el al.[585] Github–Local

function in the car package can be used to automate outlier detection and removal (it takes a model returned by glm and returns the Studentized residuals of points whose Bonferroni corrected p-value is below a cutoff threshold; default cutoff=0.05).

A method of visualizing the important influential observation and outlier information is required. The influenceIndexPlot function in the car package, takes the model returned by glm and plots the Cook's distance, Studentized residual, Bonferroni corrected p-value and hat-value for each data-point; Figure 11.14 is for the Fenton et al data.

```
all_mod=glm(KLoC ~ I(Hours^0.5), data=loc_hour)
influenceIndexPlot(all_mod, main="", col=point_col, cex.axis=0.9, cex.lab=1.0)
```

The upper plot shows four data points having a large Cook's distance, but only two of them have a significant corrected p-value (second plot from the bottom). These two data points were removed, and the process of building a model and calling influenceIndexPlot repeated; on this iteration one point is removed and iterating again finds no other data points as worthwhile candidates for removal.

Figure 11.15 shows the results of removing data points having both a high Cook's distance, and Studentized residuals whose corrected p-value is below the specified limit.

Outliers are loners, appearing randomly scattered within a plot. When multiple points appear to be following a different pattern than the rest of the data, the reason for this may be a different process driving behavior, or a change of behavior in what went before.

A study by Alemzadeh, Iyer, Kalbarczyk and Raman[29] investigated safety-critical computer failures in medical devices between 2006 and 2011 (as reported by the US Food and Drug Administration). Figure 11.16 shows the number of devices recalled for computer related problems (20-30% of all recalls), binned by two-week intervals.

Data points that stand out in figure 11.16 are the two large recall rates in the middle of the measurement interval, and recall rates at later dates appearing to increase faster than earlier; adding a loess fit (yellow) shows peaks around the two suspicious periods. The fitted straight line shows a distinct upward trend. Is this fitted line being overly influenced by the two middle period points or end of measurement period recall rates?

The measurement points appear in regular time slots, and deleting one of these time slots does not make sense; replacing an outlier with the mean of all measurements is one solution for handling this situation. Doing this (see Github–regression/Alemzadeh-Recalls.R) finds there is little change in the fitted regression model, i.e., these two outliers had little influence. Did a substantive change in the processes driving recalls, or recording of recalls, occur around the start of 2011? Further investigation, or domain knowledge, is needed to answer this question.

Figure 11.17 shows two fitted models, one using data up until the end of 2010 and the other using the data after 2010. This illustrates that blindly fitting a straight line to a sample can produce a misleading model. A change in reporting appears to have occurred around the end of 2010, which had a significant impact on reported recalls (work is needed to uncover the reason for this change) and fitting data up to the end of 2010 shows a much smaller increase, and perhaps even no increase, in recall rates, compared to when measurements after this date are included.

A change-point analysis of this data is discussed in section 11.2.9.

When combining results from multiple studies, it is possible for an entire study to be an outlier, relative to the other related studies.

A study by Amiri and Padmanabhuni[50] analysed the methods used by eleven other studies to convert between two common methods of counting function points.[xiv] Many studies included in the analysis have small sample sizes, include both student and commercial projects, and the function points are sometimes counted by academics rather than industrial developers.

Figure 11.18 shows function points counted using the COSMIC and FPA algorithms (counts made by students have been excluded). Both lines are loess fits, with red used for industry points and blue for academic researchers; the academic line overlays the industry line if one sample (i.e., Cuadtado_2007) is excluded.

The impact of influential observations on a fitted model can vary enormously, depending on the form on the equation being fitted. Figure 11.19 shows the lines of five separate

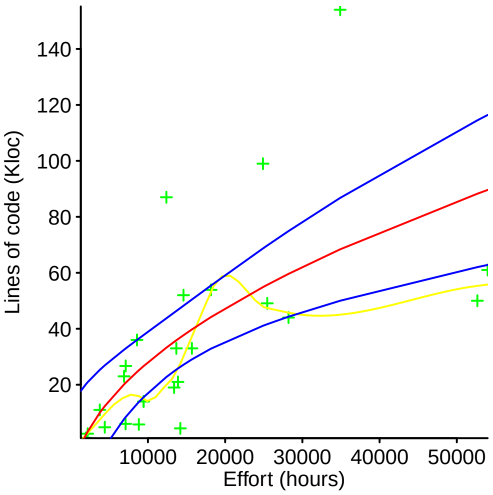


Figure 11.15: Points remaining after removal of overly influential observations, also taking into account the Bonferroni p-value of the Studentized residuals; the line shows the fitted model and 95% confidence interval (loess fit in yellow). Data from Fenton el al.[585] Github–Local

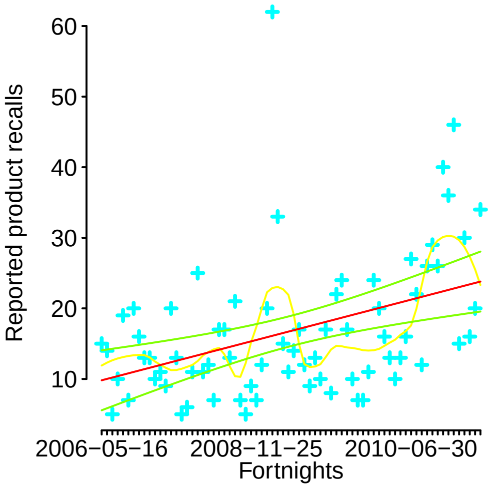


Figure 11.16: Number of medical devices reported recalled by the US Food and Drug Administration, in two week bins; fitted straight line and confidence bounds, with loess fit (yellow). Data from Alemzadeh et al.[29] Github–Local

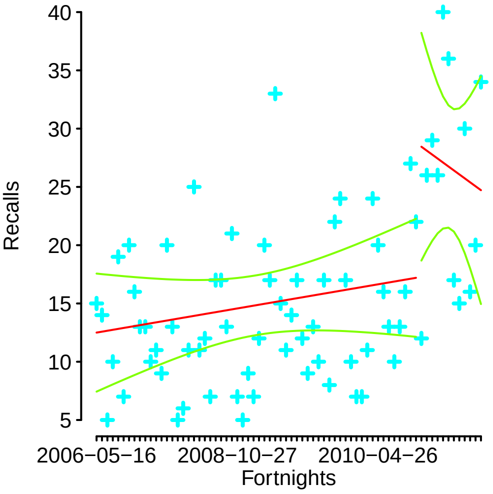


Figure 11.17: Two fitted straight lines and confidence intervals, one up to the end of 2010 and one after 2010. Data from Alemzadeh et al.[29] Github–Local

[xiv] Function point counting is a technique for estimating development effort by counting the functionality contained in the software requirements specification.

equations fitted to the Embedded subset of the COCOMO 81[212] data, with the upper plot using the original data, and the lower plot the data after three influential observations have been removed.

In some cases outlier removal has had little impact on the model fitted, while in other cases there has been dramatic changes in the coefficients of the fitted model.

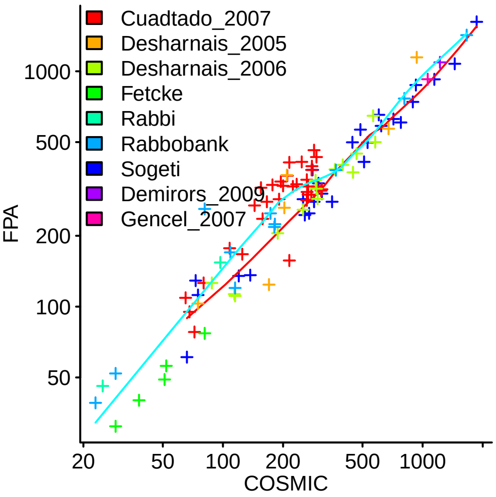


Figure 11.18: Results from various studies of software requirements function points counted using COSMIC and FPA; lines are loess fits to studies based on industry and academic counters. Data from Amiri et al.[50] Github–Local

11.2.7 Diagnosing problems in a regression model

The commonly used regression modeling functions are capable of fitting a model to almost any sample, without reporting an error (some functions are so user-friendly they gracefully handle data that produces a singular matrix, an error that is traditionally flagged, because it suggests that something somewhere is wrong). It is the analysts' responsibility to diagnose any problems in the model returned.

Looking at figure 11.20, it is visually obvious that at least two of the fitted regression lines completely fail to capture the pattern present in the data. The data set is famous, it is known as the Anscombe quartet.[62] The four samples each contain two variables, with each sample having the same mean, standard deviation, Pearson correlation coefficient and are fitted using linear regression to produce a straight line having the same slope and intercept.

Problems with a regression model are not always as obvious as the Anscombe quartet case, and diagnosing the cause of the problem can be difficult. As always, domain knowledge is very useful for suggesting alternative models or possible changes to a fitted model.

The difference between the measured value of the response variable, and the value predicted by a fitted model is known as the *residual*. While many model diagnosis techniques are based on the use of the residual, they often require more knowledge of the mathematics of regression modeling than is covered in this book.[xv]

The suggested model diagnostic techniques, for casual users of statistics, are visualization based.

Figure 11.21, upper plot, shows the residual of the straight line fitted to the Linux kernel growth data analysed in figure 11.7. Ideally the residual is randomly scattered around zero, and the V-shape seen in this plot is typical of a straight line fitted to values that curve around it (the smallest residual is in the center, where the model fits best, and is greatest at the edges; the smaller peak is a localised change of behavior, and may explain why a cubic produces a slightly better fit). This plot is one of the four diagnostic visualizations produced by `plot`, when it is passed a regression model, as follows:

```
m1=glm(LOC ~ Number_days, data=latest_version)
plot(m1, which=1, caption="", col=point_col)
```

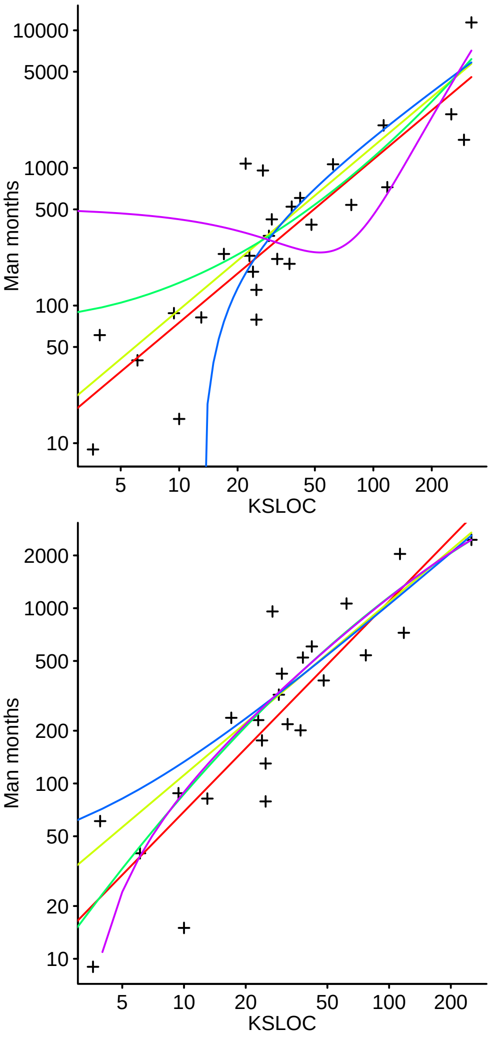


Figure 11.19: Five different equations fitted to the Embedded subset of the COCOMO 81 data before influential observation removal (upper) and after influential observation removal (lower). Data from Boehm.[212] Github–Local

Figure 11.21, lower plot, shows the original data, straight line fit (red) and loess fit (blue). Both the residual plot and loess fit express the same pattern of curvature around about the straight line fit. Both visualizations have their advantage, the loess line can be drawn before any model is fitted, while details are easier to extract, from a residual plot (e.g., values for the size of the difference).

The mathematics behind linear regression requires that each measurement be independent of all the other measurements in a sample. A common form of dependence between measurements is serial correlation, i.e., correlation between successive measurements. A fitted regression model can be tested for serial correlation using the Durbin Watson test, performed by the `durbinWatsonTest` function, in the `car` package.

A study by Flater and Guthrie[606] measured the time taken to assign a value to an array element in C and C++ , using twelve different techniques, some of which checked that the assignment was within the defined bounds of the array (two array sizes were used, large and small); the programs benchmarked were compiled using seven different compiler optimization options.

Figure 11.22 shows the timings from 2,000 executions of one technique for assigning to an array element, compiled using gcc with the 00 option (upper) and 03 option (lower). The results for 00 show a clustering of execution times for groups of successive measurements.

[xv] It is not obvious that the cost/benefit of learning the necessary mathematics is worthwhile (but it is a good source of homework exercises for students).

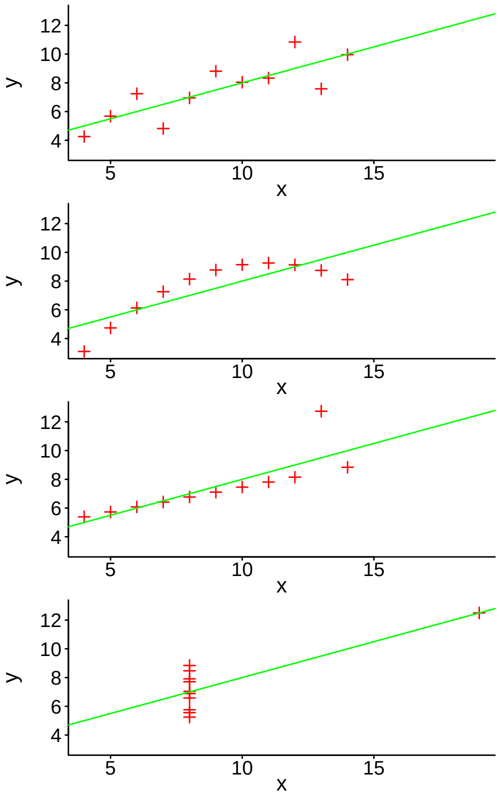


Figure 11.20: Anscombe data sets with Pearson correlation coefficient, mean, standard deviation, and line fitted using linear regression. Data from Anscombe.[62] Github–Local

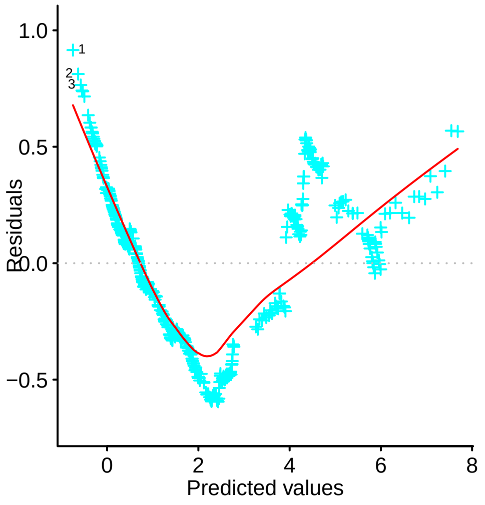


Figure 11.21: Residual of the straight line fit to the Linux growth data analysed in figure 11.7. Data from Israeli et al.[891] Github–Local

A Durbin Watson test confirms that the 00 measurements are correlated (see Github–benchmark/array-durbanwatson.R).

Some regression modeling functions can adjust for the presence of serial correlation (information about the correlation is passed in an optional argument). The `gls` function, in the `nlme` package, supports a `correlation` option; the `dynlm` package supports the use of time series operators (e.g., diff and lag) in the specification of model formula; the `tscount` package supports the fitting of generalized linear models to time series of count data.

When measurements contain a significant amount of serial correlation, time-series analysis techniques may provide useful information; see section 11.10. The `tsglm` function, in the `tscount` package, supports regression modeling of count time series.

11.2.8 A model's goodness of fit

How well does a model fit the data? The term *goodness of fit* is often used to describe this quantity. Various formula for calculating a goodness of fit have been proposed, and often involve the difference between the value measured and the corresponding value predicted by the model.

For the end-user of results of the analysis, meeting expectations of behavior is an important model characteristic.

When fitting equations to gain understanding, the structure of the processes suggested by the terms of the equation are an important characteristic.

When making predictions, the primary quantity of interest is the accuracy of new predictions, i.e., the amount of expected error in predictions for values that are not in the sample used to build the model. The error structure is also a consideration; is the priority to minimise total error, worse case error, to prefer over-estimates to under-estimates (or vice versa) or does some complicated weighting (over the range of values that explanatory variable(s) might take) have to be taken into account?

When dealing with one explanatory variable, it is possible to get a good idea of how well a model fits the data through visualization, e.g., by plotting them both. Does the fitted line look correct and how wide are the confidence intervals? However, for data containing more than one explanatory variable, accurate visualizations are problematic.

To create a model by fitting it to data, is to create a just so story. The predictions made by a model, outside the range of the data used to build it, are just something to discuss when considering expectations of behavior (which might be derived from a theory of the processes involved in generating the data used to fit the model).

Confidence intervals, see fig 11.3, provide information about the goodness of fit at every point. The following discussion looks at some ways of producing a single numeric value, to represent goodness of fit.

The leftover variation in a sample that is not accounted for by the fitted model, the residual, is invariably a component in any equation in the calculation of a single value to summarise how well the model performs. Some of the equations used include:

- `null deviance` is a measure of the difference between the data and the mean of the data, `deviance` is a measure of the difference between the data and a fitted model (both values are listed in the `summary` output of a model fitted by `glm`). The percentage difference between the deviance and null deviance is a measure of the variance in the data that is not explained by the mode,
- R-squared (also known as the *coefficient of determination* and commonly written R^2) can be interpreted as the amount of variance in the data (as measured by the residuals) that is explained by a model. It takes values between zero and one (which has the advantage of being scale invariant) and is a measure of correlation, not accuracy.

  Sometimes the adjusted R^2, written $\bar{R}^2$, is used, which takes into account the number of explanatory variables, p, and sample size, n: $\bar{R}^2 = R^2 - (1 - R^2)\frac{p-1}{n-p}$
- mean squared error (MSE): the mean squared error is the mean value of the square of the residuals and as such has no upper bound (and will be heavily influenced by outliers); root mean squared error (RMSE) is the square-root of MSE.

  The following equation shows how MSE and R^2 are related: $R^2 = 1 - \frac{MSE}{\sigma^2}$

- mean absolute error (MAE): the mean absolute error is the mean value of the absolute value of the residuals. This measure is more robust in the presence of outliers than MSE.

Apart from the R^2 metric, the metrics listed (plus AIC) are scale dependent, e.g., mapping measurements from centimeters to inches changes their value; transforming the scale (e.g., taking logs) will also change metric values.

The choice of a metric is driven by what information is available and what model characteristics are considered important (e.g., how important is being able to handle outlier). In a competitive situation, people might not be willing to reveal details about their model and so any public metric has to be based on predictive accuracy (e.g., model builders provide the predictions made by their model to a test data set).

R^2 is the only scale invariant metric, and it provides an indication of how much improvement might be possible over an existing model.

It is possible for the coefficients of a fitted model to be known with a high degree of accuracy, and yet for this model to explain very little of the variance present in the data, and for there to appear to be little chance of improving on the model given the available data.

The Ultimate Debian Database project[1842] collects information about packages included in the Debian Linux distribution. Figure 11.23 shows the age of a packaged application plotted against the number of systems on which that application is installed, for 14,565 applications in the "wheezy" version of Debian; also, see fig 6.4 and fig 8.18.

The fitted linear model (red line, hidden by the 95% confidence interval in green overwriting it; loess fit in blue) has a very low p-value, a consequence of the large number of, and uniform distribution of, data points. The predictive accuracy of this model is almost non-existent, the only information it contains is that older packages are a little more likely to be installed that younger ones.

A study by Jørgensen and Sjøberg[946] investigated developers' ability to predict whether any major unexpected problems would occur during a software maintenance task. Building a regression model, using the available measured attributes, finds that lines of code is the only explanatory variable having a p-value less than 0.05. However, only 3.3% of the variance in the response variable is explained by the number of lines of code; while the explanatory variable was statistically significant, its practical significance was negligible (see Github–maintenance/10.1.1.37.38.R).

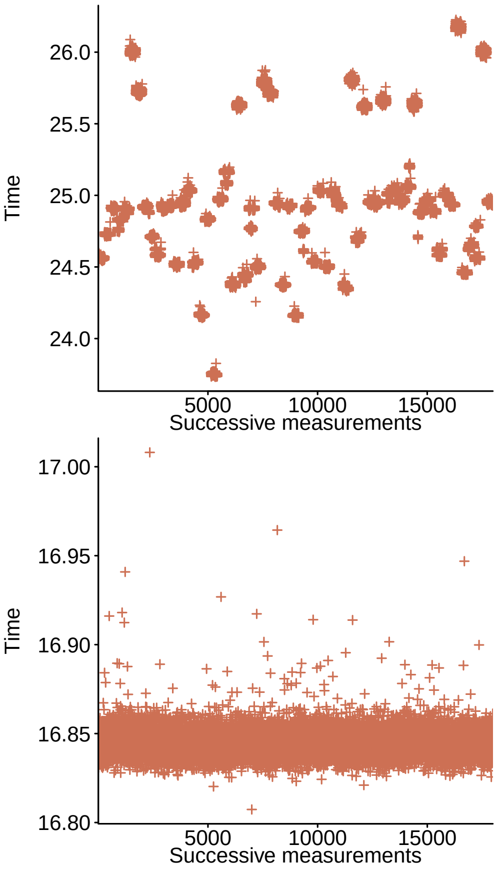


Figure 11.22: Array element assignment benchmark compiled with gcc using the O0 (upper) and O3 (lower) options (measurements were grouped into runs of 2,000 executions). Data from Flater et al.[606] Github–Local

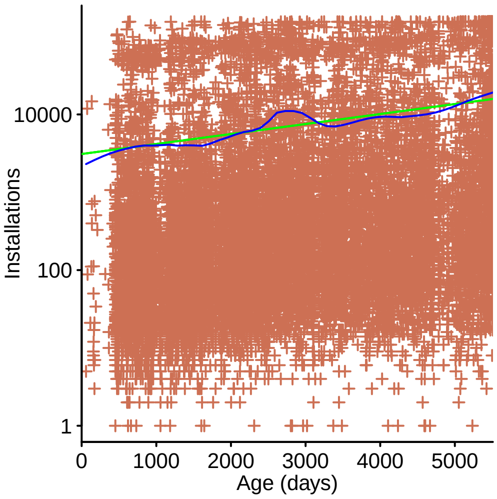


Figure 11.23: Number of installations of Debian packages against the age of the package, plus fitted model and loess fit. Data from the "wheezy" version of the Ultimate Debian Database project.[1842] Github–Local

11.2.9 Abrupt changes in a sequence of values

When the processes generating the measured values change, the statistical properties of the post-change sequence of values may abruptly change. The point where the statistical properties of a sequence of values significantly changes is known as a *change-point*.

The changepoint package supports basic change-point analysis of the mean and variance of a sequence of values. The cpt.mean function checks for significant shifts in the mean value; the method="AMOC" (At Most One Change) option searches for what its name implies; other values support searching for a specified maximum number of changes, with method="PELT" selecting what is considered to be the optimum number of changes.

An earlier analysis of electronic device recalls (see fig 11.17) suggested that a significant shift in the processes driving reported recalls occurred at the end of 2010. Figure 11.24 shows the output from the following calls to cpt.man:

```
library("changepoint")

change_at=cpt.mean(as.vector(t2))
plot(change_at, col=point_col,
        xlab="", ylab="Reported product recalls\n")

change_at=cpt.mean(as.vector(t2), method="PELT")
plot(change_at, col=point_col,
        xlab="Fortnights", ylab="Reported product recalls\n")
```

For an example of detecting changes in variance and changes in both mean and variance, see Github–regression/hpc-read-write.R.

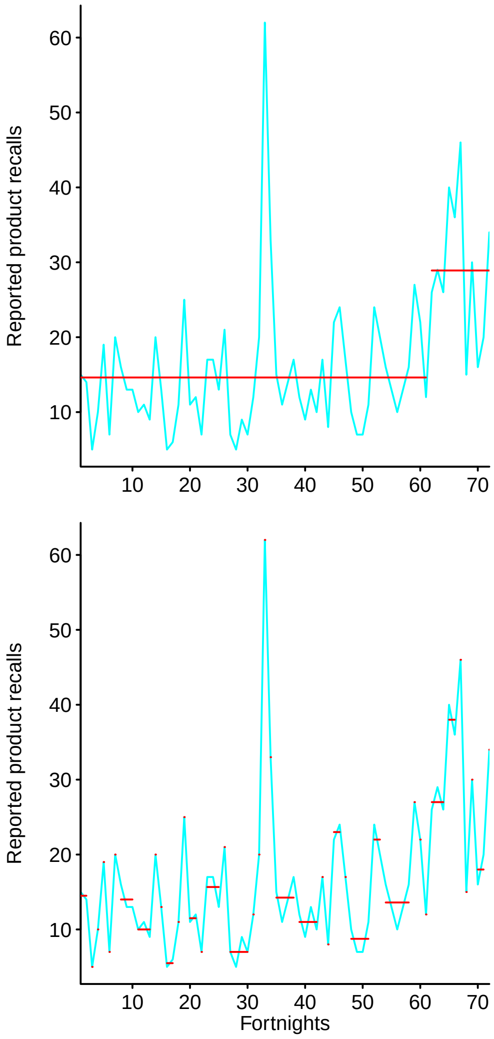


Figure 11.24: Change-points detected by `cpt.mean`, upper using `method="AMOC"` and lower using `method="PELT"`. Data from Alemzadeh et al.[29] Github–Local

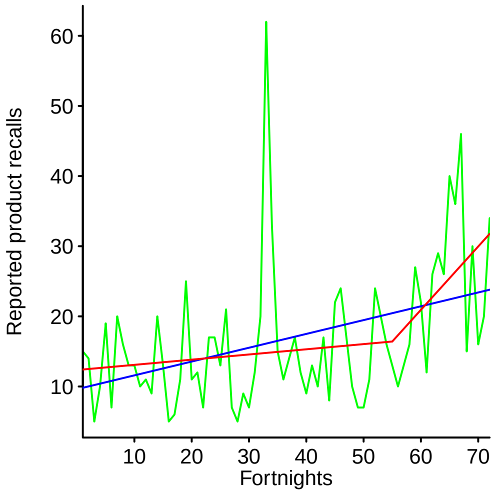


Figure 11.25: Fitted regression model (blue) and adjusted model with one change-point (red). Data from Alemzadeh et al.[29] Github–Local

The `segmented` function in the `segmented` package adjusts a fitted regression model to take account of change-points; at the time of writing this function only fits connected line segments, i.e., no disjoint line boundaries, such as that present in figure 11.17. A model is fitted using `glm` is passed to `segmented`, which attempts to estimate the appropriate change points and fit a series of line segments between each change-point (the number of change-points can be explicitly specified). Figure 11.25 shows the output from the following code (also see fig 10.20):

```
library("segmented")

plot(t2$fortnight, t2$freq, type="l", col=pal_col[2], xaxs="i",
        xlab="Fortnights", ylab="Recalls\n")

al_mod=glm(freq ~ fortnight, data=t2) # fit model as usual

pred=predict(al_mod)
lines(pred, col=pal_col[3]) # add fitted line to plot

seg_mod=segmented(al_mod, npsi=1) # adjust fitted model with one change-point

plot(seg_mod, col=pal_col[1], add=TRUE) # add fitted lines to plot
```

When the location of the change-point is known, or the `segmented` function fails to find a reasonable fit, an abrupt change can be modeled using `glm` to effectively fit multiple equations; one equation over each discontinuity separated interval. While each equation may be fitted by an independent call to `glm`, it may be possible to build a single model incorporating every discontinuity.

Figure 11.26 shows an abrupt change in the sales volume of 4-bit microprocessors (green). Straight lines have been fitted to the two periods before/after April 1998 (red), with the yearly sales cycle modeled by a sine wave (blue).

The technique for fitting a model that handles discontinuous patterns of behavior makes use of an interaction between the explanatory variable (`date` in this case), and a dummy variable whose 0/1 value depends on `date`, relative to the change-point. The code for the straight line model (red line) is:

```
y_1998=as.Date("01-04-1998", format="%d-%m-%Y")  # estimated discontinuity point

p4=glm(bit.4 ~ date*(date < y_1998)+date*(date >= y_1998), data=proc_sales)
```

and the `summary` output is: Github–Local

```
Call:
glm(formula = bit.4 ~ date * (date < y_1998) + date * (date >=
    y_1998), data = proc_sales)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-19756.9   -6372.8    -558.7    6533.2   19086.4

Coefficients: (2 not defined because of singularities)
                          Estimate Std. Error t value Pr(>|t|)
(Intercept)              7.050e+04  5.802e+04   1.215   0.2265
date                     7.072e-01  5.368e+00   0.132   0.8954
date < y_1998TRUE       -8.357e+04  5.873e+04  -1.423   0.1572
date >= y_1998TRUE              NA         NA      NA       NA
date:date < y_1998TRUE   1.045e+01  5.466e+00   1.912   0.0581 .
date:date >= y_1998TRUE         NA         NA      NA       NA
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 80003408)

    Null deviance: 2.0763e+10  on 131  degrees of freedom
Residual deviance: 1.0240e+10  on 128  degrees of freedom
AIC: 2782.6

Number of Fisher Scoring iterations: 2
```

Sales follow a seasonal trend that can be approximated using a sine wave having a 12-month frequency, adding this to the straight line model as follows:

```
season_p4=glm(bit.4 ~ date*(date < y_1998)+date*(date >= y_1998)+
                                 sin(rad_days)+cos(rad_days), data=proc_sales)
```

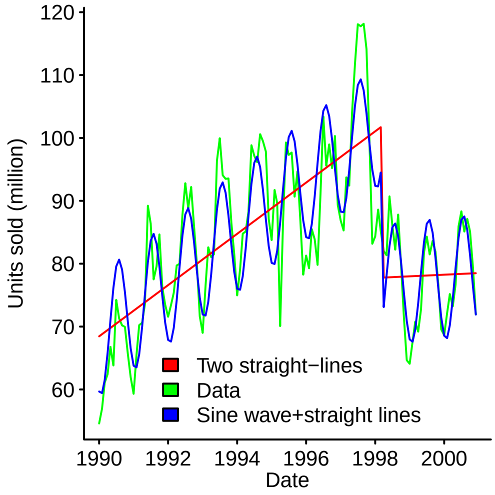


Figure 11.26: Monthly unit sales (in millions) of 4-bit microprocessors. Data kindly supplied by Turley.[1833] Github–Local

11.2.10 Low signal-to-noise ratio

Measurements sometimes contain a large amount of noise, relative to the signal present, i.e., a low signal-to-noise ratio. Fitting a model to such data can be difficult, because many equations do an equally (not very) good job.

The two plots along the upper row in figure 11.27 show data generated from a quadratic equation containing noise, along with two fitted models (red and blue lines). The equation used to generate the two sets of data is:

$$y = x^2 + K \times (5 + rnorm(length(x)))$$

where: $K = 10^3$ (left column), and $K = 10^2$ (right column).

It is not possible to tell by looking at the upper left plot whether a quadratic (blue), or an exponential (red), is a better fit; the output from `summary` is not much help (see Github–regression/noisey-data.R). The upper right plot contains less noise, and it is easier to see that the exponential fit does not follow the data as well as the quadratic.

Sometimes the peaks (or troughs) in the plotted data can be an indicator of the shape of the data. The upper left plot includes a quadratic and exponential fit to the three largest values at each x-value (the fitted model does not seem to have less the uncertainty in this case).

The *ratio test* is a technique that can help rule out some equations as possible candidates for modeling. If $f(x)$ is the function being fitted to the data and this data was generated by the function $g(x)$, the ratio $\frac{g(x)}{f(x)}$ will converge to a constant as x becomes small/large enough such that the signal dominates the noise.

The two plots along the lower row in figure 11.27, show ratio tests for quadratic (blue), cubic (red) and exponential (green) equations. The exponential equation shows no sign of converging to a constant, while quadratic is closer to doing this than cubic (which can be ruled out because it does a poor job of fitting the data).

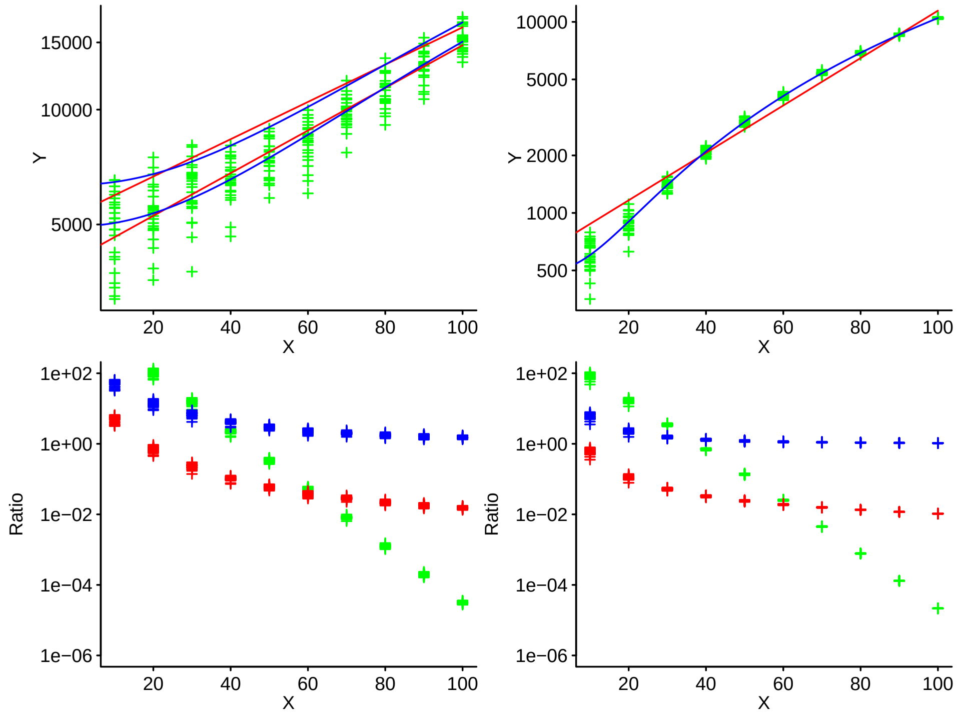


Figure 11.27: Quadratic relationship with various amounts of added noise, fitted using a quadratic and exponential model. Github–Local

The ratio test rules out an exponential equation being a good candidate for fitting a model to the data in figure 11.27.

A study by Vasilescu, Serebrenik, Goeminne and Mens[1865] investigated contributions to the Gnome ecosystem, from the point of view of workload (measured as the number of file touches, e.g., commits), breaking it down by projects, authors and number of activity types (e.g., coding, testing, documentation, etc).

Figure 11.28, upper plot, shows, for individual authors, workload and the number of activity types they engaged in. There is a large amount of noise in the data (or variance not explained by the explanatory variable used for the x-axis). Figure 11.28, lower plot, shows a ratio test, with an exponential failing to level off, the linear equation slowly growing, and the quadratic looking like it is trying to grow.

Perhaps the behavior would become clearer with more activity types, but the quadratic is the only candidate not ruled out.

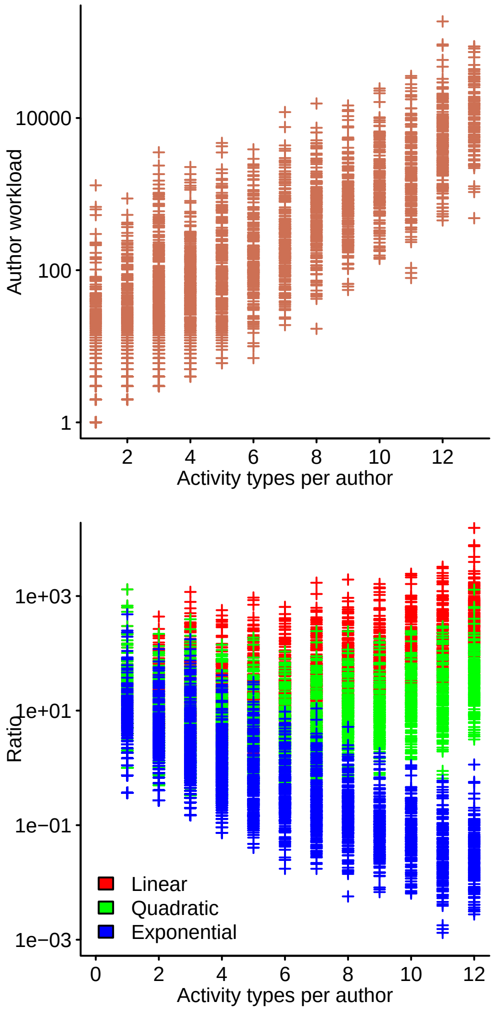


Figure 11.28: Author workload against number of activity types per author (upper) and ratio test (lower). Data from Vasilescu et al.[1865] Github–Local

11.3 Moving beyond the default Normal error

Measurements sometimes have properties that do not meet the requirements assumed by the mathematics on which glms default argument values are based. Some measurement properties that non-default argument values can handle, include the response variable having values that:

- can never go below zero, e.g., count data,
- can never be greater than some maximum value, e.g., some percentages can never be greater than 100,
- span several orders of magnitude and contain an additive error.

By default, glm uses a Normal distribution for the measurement error. Figure 11.29 shows a fitted regression line with four data points (red stars adjacent to a black line); the colored Normal curves over each point represents the probability distribution of the measurement error that is assumed to have occurred for that measurement (the center of each error distribution curve is directly above the fitted line at each explanatory variable measurement point).

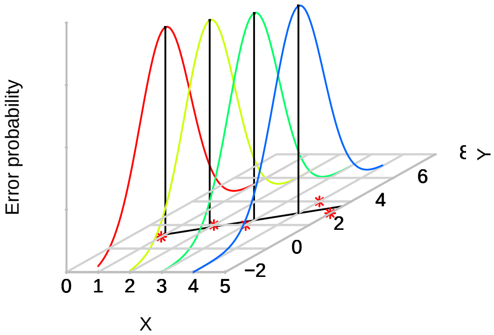


Figure 11.29: Fitted regression line to points (in red) and 3-D representation of assumed Normal distribution for measurement error. Github–Local

glm's family argument has the default value family=gaussian(link="identity"), which can be shortened to family=gaussian (the default link function for gaussian islink="identity").

The Normal distribution includes negative values and when a measurement cannot have a negative value, using an error distribution that includes negative values can distort the fitted model. One alternative is the Poisson distribution, which is zero for all negative values. The following call to glm specifies that the measurement error has a Poisson distribution:

```
a_model=glm(a_count ~ x_measure, data=some_data, family=poisson)
```

After Normal, the Poisson and Beta distributions are the most common measurement error distributions used by the analysis in this book.

Calling glm, with a non-default value for the family argument, requires knowing something about the mathematics behind generalised regression model building. The equation actually being fitted by glm is:

$$l(y+\varepsilon) = \alpha + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

which differs from the one given at the start of the chapter in having $l(y+\varepsilon)$ on the left-hand-side, rather than y. This l is known as the *link function*, which for the Normal distribution is the identity function (this leaves its argument unmodified, and the equation ends up looking like the one given at the start of this chapter).

Once a regression model is fitted, the value of the response variable is calculated from:

$$y = l^{-1}(\alpha + \beta_1 x_1 + \beta_2 x_2 + \cdots +) - \varepsilon$$

where: l^{-1} is the inverse of the link function used, e.g., the inverse of log is e raised to the appropriate power.

Every error distribution has what is known as a *canonical link* function, which is the function that pops out of the mathematical analysis for that distribution. By default, glm uses the canonical link function for each error distribution, and allows some alternatives to be specified. The canonical link function for the Poisson distribution is `log`.

When the link function is not `identity`, prediction values and confidence intervals need to be mapped as follows:

```
a_pred=predict(a_model, se.fit=TRUE)
inv_link=family(a_model)$linkinv      # get the inverse link function

lines(x_values, inv_link(a_pred$fit)) # fitted line
# confidence interval above and below
lines(x_values, inv_link(a_pred$fit+1.96*a_pred$se.fit))
lines(x_values, inv_link(a_pred$fit-1.96*a_pred$se.fit))
```

The analysis in the following sections involve measurements that require the use of a variety of measurement error distributions and link functions.

11.3.1 Count data

Count data has two defining characteristics, it is discrete and has a lower bound of zero. The discrete distribution taking on non-negative values, supported by `glm`, is the Poisson distribution.

In practice, when measurement values are sufficiently far away from zero (where far may be more than 10) there is little difference between models fitted using the Normal and Poisson distributions. For measurements close to zero, the main difference between models fitted using different distributions is the confidence intervals (which are usually not symmetric, and may be larger/smaller).

The canonical link function for the Poisson distribution is `log`, and the following two calls to `glm` are equivalent:

```
p_mod=glm(y ~ x, data=sample, family=poisson)
p_mod=glm(y ~ x, data=sample, family=poisson(link="log"))
```

The `log` link function means that the equation being fitted is actually:

$$y = e^{\alpha+\beta x} + \varepsilon$$

To fit the equation: $y = \alpha + \beta x + \varepsilon$, for a Poisson error distribution, the `identity` link function has to be used, as follows (experience shows that `glm` sometimes fails to converge when `family=poisson(link="identity")` is specified and that `start` values have to be specified):

```
p_mod=glm(y ~ x, data=sample, family=poisson(link="identity"))
```

A study of the effectiveness of security code reviews by Edmundson, Holtkamp, Rivera, Finifter, Mettler and Wagner[523] asked professional developers, with web security review experience, to locate vulnerabilities in web code. The number of vulnerabilities found can only be a non-negative integer value and in this study were single digit values.

The values fitted to a discrete distribution consists of a series of discrete steps, as the upper plot of figure 11.30 shows (fitted line and 95% confidence intervals). While this plot is technically correct, it is ambiguous: are the values specified by the top left edge, or the bottom right edge of the staircase?[xvi] Plots using continuous lines are simpler for readers to interpret and so are used in this book.

The dashed lines in figure 11.30, lower plot, were fitted using `glm`'s default values,[xvii] while the argument `family=poisson(link="identity")` was used to fit the model represented by the smooth lines.

The two fitted lines are virtually identical (the green dashed line is drawn over the continuous red line), but the 95% confidence intervals do differ. This pattern of behavior is very common, unless the response variable has many values near zero.

Is the difference between fitting a model using the technically correct Poisson distribution, or a Normal distribution, worth the effort (for the analyst, not the use of any additional computing resources)?

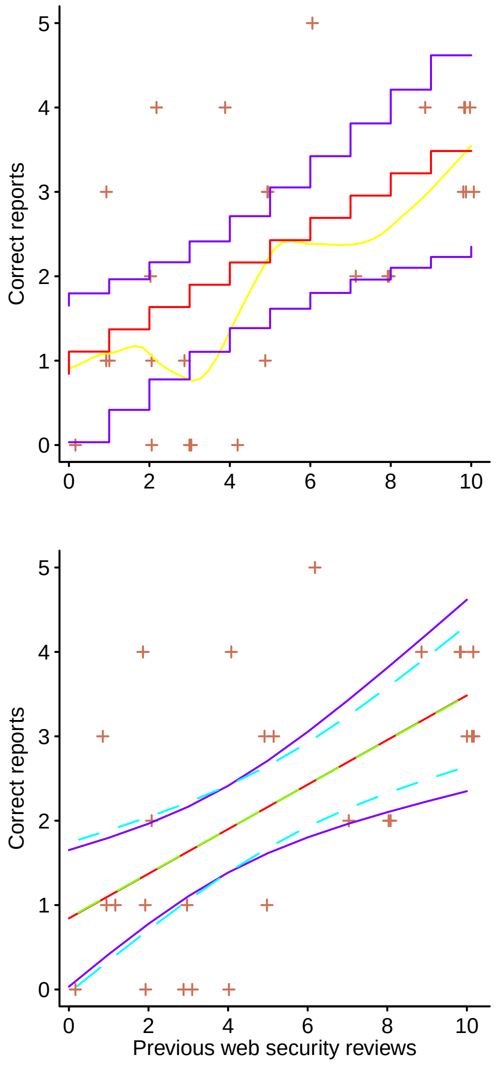


Figure 11.30: Number of vulnerabilities detected by professional developers with web security review experience; upper: technically correct plot of model fitted using a Poisson distribution, lower: simpler to interpret curve representation of fitted regression models assuming measurement error has a Poisson distribution (continuous lines), or a Normal distribution (dashed lines). Data extracted from Edmundson.[523] Github–Local

[xvi] The choice is selectable via the `type` argument to `plot/lines`.

[xvii] To achieve an acceptable p-value, three outliers were removed.

Sometimes the Poisson distribution is used because a log link function transforms the response variable, while keeping an additive measurement error.

When fitting models containing multiple explanatory variables (discussed later) and a response variable containing count data, it can be more difficult to detect differences between using a Poisson and Normal distribution. While use of the Poisson distribution may involve more effort, it removes uncertainty and is always worth trying.

The Negative Binomial distribution is perhaps the second most commonly encountered count distribution. A study by Jones[919] included counting the number of break statements in C functions. Figure 11.31 shows the number of functions containing a given number of break statements, along with a fitted Negative Binomial distribution.

A break statement can occur zero or more times within a loop or switch statement, and these statements can occur zero or more times within a function definition. A Negative Binomial distribution can be generated by drawing values from multiple Poisson distributions (whose characteristics have been drawn from a Gamma distribution); might the number of break statements in each function different Poisson distribution?

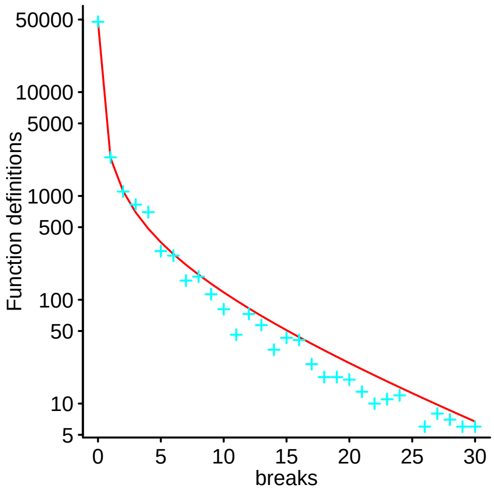


Figure 11.31: Number of functions containing a given number of break statements and a fitted Negative Binomial distribution. Data from Jones.[919] Github–Local

The gamlss package[1744] supports a wide variety of probability distributions, including the NBI distribution (Negative binomial type I distribution; there is also a type II) used in the following code:

```
library("gamlss")

breaks=rep(j_brk$occur, j_brk$breaks)
nbi_bmod=gamlss(breaks ~ 1, family=NBI)

plot(function(y) max(jumps$breaks, na.rm=TRUE)*     # Scale probability distribution
                        dNBI(y, mu=exp(coef(nbi_bmod, what="mu")),
                                sigma=exp(coef(nbi_bmod, what="sigma"))),
        from=0, to=30, n=30+1, log="y", col=pal_col[1],
        xlab="breaks", ylab="Function definitions\n")
points(jumps$occur, jumps$breaks, col=pal_col[2])
```

While zero is a common lower bound, other lower bounds are sometimes encountered; see section 9.3.1. Both the gamlss.tr and VGAM packages support a wide variety of truncated distributions; gamlss and related packages are used in this book because of the volume and quality of their documentation.

A study by Starek[1741] investigated API usage in Java programs. Figure 11.32 shows the number of APIs used in Java programs containing a given number of lines of code. The API count starts at one, not zero, and many programs use a few APIs, suggesting that a Poisson distribution may be applicable; the range of the number of APIs used does not suggest a log scale.

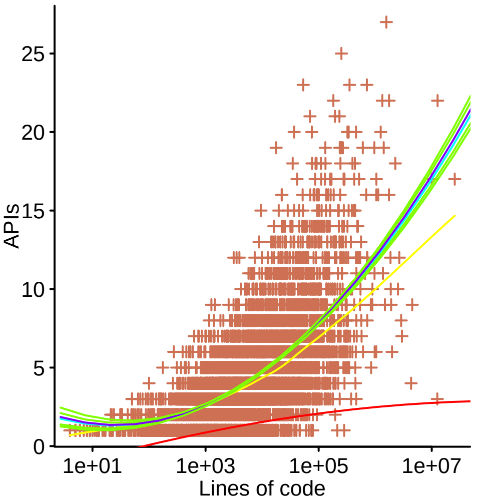


Figure 11.32: Number of APIs used in Java programs containing a given number of LOC; lines are fitted models based on a zero-truncated Poisson (red), Poisson and Normal distributions, yellow line is loess fit. Data from Starek.[1741] Github–Local

In the following code, gen.trun creates a zero-truncated Poisson distribution (derived from PO in the gamlss.tr package) having the identity function as the link for its mean (rather than the default log link).

```
library("gamlss")
library("gamlss.tr")

gen.trun(par=0, family=PO(mu.link=identity))

tr_mod=gamlss(APIs ~ l_size+I(l_size^2), data=API, family=POtr)
```

Figure 11.32 shows the fitted model in red. The other lines are fitted models using a Poisson distribution that is not zero-truncated and a Normal distribution, along with 95% confidence intervals. The yellow line is a loess fit. Other explanatory variables could be added to the model to improve the fit to the data.

11.3.2 Continuous response variable having a lower bound

Measurements of the response variable may be drawn from a continuous distribution, e.g., measurements involving length or time. The continuous distribution taking non-negative values, supported by glm, is the Gamma distribution.

In practice, when most measurement values are sufficiently far away from zero (where far away could be a large single digit value) there is little difference between models fitted using the Normal and Gamma distributions. For measurements closer to zero the main difference between models fitted using different distributions is in the confidence intervals (which are usually not symmetric, and may be larger/smaller).

The canonical link function for the Gamma distribution is `inverse`, and the following two calls are equivalent:

```
G_mod=glm(y ~ x, data=sample, family=Gamma)  # Yes, capital G
G_mod=glm(y ~ x, data=sample, family=Gamma(link="inverse"))
```

The `inverse` link function means that the equation being fitted is (the `identity` link function is supported):

$$y = \frac{1}{\alpha + \beta x} + \varepsilon$$

Figure 11.33 comes from a code review study (discussed in section 13.2) and shows meeting duration when reviewing various amounts of code. Meeting duration must be greater than zero, and a Gamma measurement error distribution is assumed to apply (the variables are assumed to have a linear relationship, and the identity link function is used). The red line is the model fitted using a Gamma error distribution (plus confidence bounds), the green line is the Gaussian distribution fit.

The data contains a few points with high leverage, and the loess fit suggests that there may be a change-point, so a more involved analysis is appears necessary.

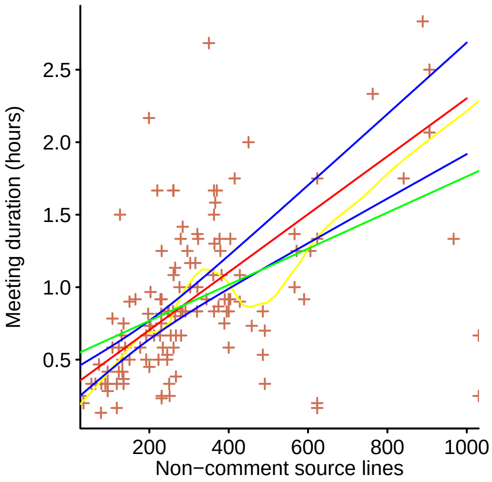


Figure 11.33: Code review meeting duration for a given number of non-comment lines of code; fitted regression model, assuming errors have a Gamma distribution (red, with confidence interval in blue), or a Normal distribution (green). Data from Porter et al.[1487] Github–Local

11.3.3 Transforming the response variable

When plotting sample points, values along one or both axes are sometimes transformed to compress or spread out the points, for the purpose of improving data visualization.

A regression model is fitted to a pattern (represented by an equation) and if a plot using transformed axis, contains visible pattern(s) of behavior, it is worth investigating a model that uses similarly transformed values.

Applying a non-linear transform to the response variable changes its error distribution, and a regression model built using this transformed response variable may not be a natural fit to the processes that produced the measurements. Explanatory variables are assumed not to contain any error and transforming them does not change this assumption.

For example, in the following regression model the error, ε, is additive:

$$y = \alpha + \beta x + \varepsilon$$

while fitting a log-transformed response variable:

$$\log y = \alpha + \beta x + \varepsilon$$

produces a model where the error is multiplicative, i.e., the error is a percentage of the measured value:

$$y = e^{\alpha + \beta x} e^{\varepsilon}$$

The error in a model fitted using a log link function is additive, because the equation fitted is:

$$\log(y + \varepsilon) = \alpha + \beta x$$

which becomes (the error randomly fluctuates around zero, and negating it changes nothing):

$$y = e^{\alpha + \beta x} + \varepsilon$$

If the response variable is transformed, the decision on whether to transform it directly, or via a link function, is driven by whether the error is thought to be additive or multiplicative; as always, domain knowledge is crucial.

A log link can be specified for `glm`'s default Normal distribution by passing: `family=gaussian(link="log")`. If this use fails to converge, the Poisson distribution is a good approximation to the Normal distribution (except when many sample values are close to zero) and can be substituted when the response variable takes integer values (see fig 7.34).

One advantage of log transforming a response variable is that it reduces the influence of outliers (because the range of values is compressed). Figure 11.34 illustrates the impact of removing one highly influential value (circled in red) from the data used to fit a model using a log link function (blue lines, dashed is after removal), and a model fitted using a log transformed response variable (red lines, dashed is after removal);[xviii] the vertical shift is the difference between treating measurement error as additive and multiplicative.

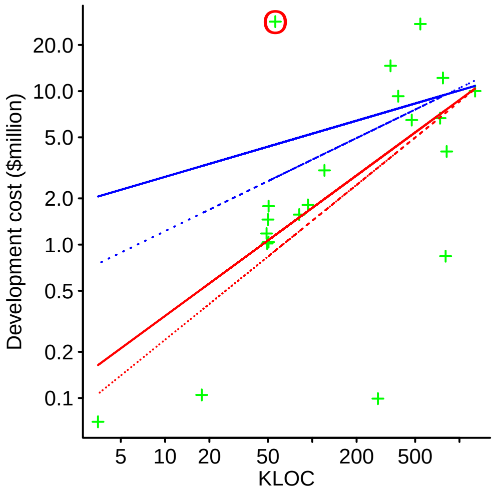


Figure 11.34: Annual development cost and lines of Fortran code delivered to the US Air Force between 1962 and 1984; lines show fitted regression models (red: log transformed, blue: using a log link function) before(solid)/after(dotted) outlier removed (circled in red). Data extracted from NeSmith.[1346] Github–Local

The visual appearance of outliers and influential observations plotted using log axis can be deceiving, i.e., they may not appear to be that far removed from the general trend (see fig 8.40). As always, assumptions based on visual appearance need to be checked numerically.

Many data analysts continue to fixate on fitting data whose measurement error has a Normal distribution. The Box-Cox transformation continues to be used to map a response variable to have a more Normal distribution-like error. The boxcox function in the MASS package, and the powerTransform function in the car package, provide support for this functionality.

A traditional approach to simplifying a problem is to map a continuous variable to a number of discrete values (e.g., small/medium/large). Throwing away information may simplify a problem, but the cost can be a considerable loss of statistical power and residual confounding.[1593] Using a computer removes the need to simplify just to reduce the manual effort needed to perform the analysis. See Github–regression/melton-statics.R for an example where building a regression model provides a lot more information about the characteristics of the continuous data, compared to mapping values to large/small and running a chi-squared test.

Adjusting a fitted model to handle uncertainty in the explanatory variables, when the model contains a multiplicative error, requires specifying the measurement error for every value of an explanatory variable. The following option assigns a 10% error: measurement.error=maint$lins_up/10.

A study by Jørgensen[933] investigated maintenance tasks and obtained developer effort and code change data. Figure 11.35 shows the effort (in days) and number of lines inserted and updated for 89 maintenance tasks. The original fitted regression line is in red, and the SIMEX adjusted line is in blue. The call to simex is:

```
maint_mod=glm(EFFORT ~ lins_up, data=maint,
                        family=gaussian(link="log"), x=TRUE, y=TRUE)

y_err=simex(maint_mod, SIMEXvariable="lins_up",
              measurement.error=maint$lins_up/10, asymptotic=FALSE)
```

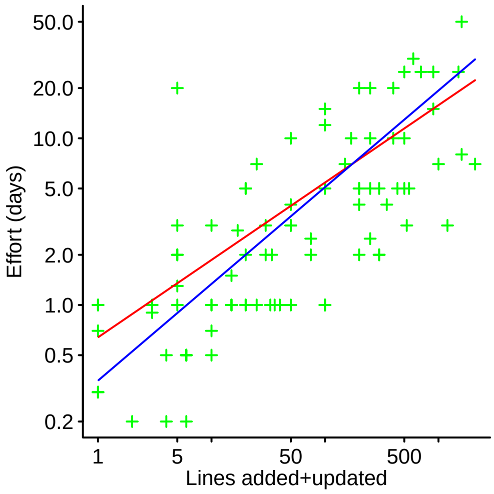


Figure 11.35: Maintenance task effort and lines of code added+updated, with fitted regression model (red), and SIMEX adjusted for estimated 10% error (blue). Data from Jørgensen.[933] Github–Local

11.3.4 Binary response variable

When the response variable takes one of two possible values, e.g., (false, true) or (0, 1), it has a binomial distribution. If the value of the response variable switches from 0 to 1 (or 1 to 0), as the explanatory variable increases (or decreases), and then always has that value for further increases (decreases) in the explanatory variable, there is no need to build a regression model (simply find the switch point). When the response variable can have two possible values over some range of the explanatory variable, regression modeling fits an equation that minimises some metric for the residual error.

A study by Höfer[828] investigated the various aspects of the implementation a problem by students and professional developers (working in pairs). Figure 11.36 shows the number of lines of test code changed by students and professionals (measurement denoted by the grey plus is treated as an outlier and not included in the model building), along with fitted regression lines, a straight line and a logistic equation. (For an analysis of Microsoft's C/C++ compiler price differential under MSDOS and Windows, see Github–economics/upgrade-languages.R).

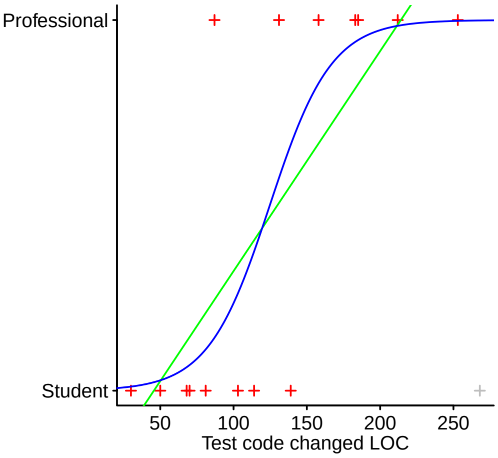


Figure 11.36: Regression modeling 0/1 data with a straight line and a logistic equation. Github–Local

The canonical link function for the Binomial distribution is logit, and the following two calls are equivalent:

```
b_mod=glm(y ~ x, data=sample, family=binomial)
b_mod=glm(y ~ x, data=sample, family=binomial(link="logit"))
```

The equation for the logit link function is:

[xviii] The visual difference is less dramatic if the axes are switched.

$$\log \frac{y}{1-y} = \alpha + \beta x$$

where the response has the form of a log-odds ratio. This equation can also be written as:

$\log \frac{p}{q} = \alpha + \beta x$, where: p proportion of successes, q proportion of failures ($q = 1 - p$).

$$p = \frac{e^{\alpha+\beta x}}{1+e^{\alpha+\beta x}}$$

The values returned by `predict`, for a fitted binomial model, are in the range $0 \ldots 1$. The person doing the analysis has to decide the value that divides this continuous range, such that predictions are either zero or one. One approach is to treat predicted values greater than 0.5 as predicting one, and predicted values less than or equal to 0.5 as predicting zero. A more sophisticated approach looks at the distribution of predictions, and makes an informed trade-off between the true/false positive rate (often calculated using *recall* and *precision*).

A ROC curve (receiver operating characteristics; named after a technique originally used to measure the performance of radio receivers) is a visualization technique showing the trade-offs between two rates, i.e., the true positive rate and false positive rate; it is a common technique for displaying the trade-offs from predictions returned by machine learning models. The `ROCR` package supports the creation and plotting of ROC curves.

The columns in table 11.1 show an example of the impact of selecting particular cut-off values, for distinguishing between true/false (for 10 data points). Reading left-to-right, at a cut-point of 0.9 there is one correct prediction (a true positive), at a 0.81 cut-point another correct prediction, while at 0.72 an incorrect prediction (a false positive) is made (at this cut-point the response rate for correct predictions is 40% and 20% for incorrect predictions).

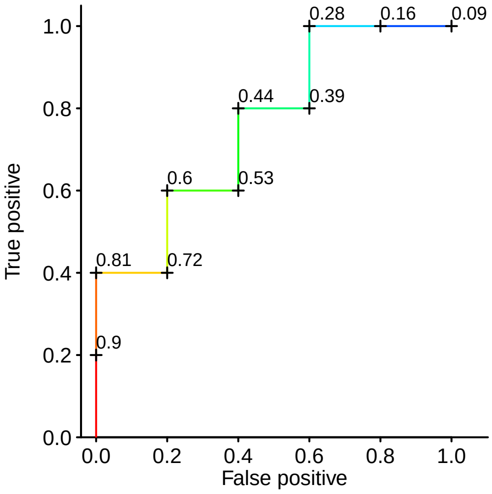


Figure 11.37: ROC curve for the data listed in table 11.1. Github–Local

t	t	f	t	f	t	f	t	f	f
0.90	0.81	0.72	0.60	0.53	0.44	0.39	0.28	0.16	0.09

Table 11.1: Example list of prediction outcome occurring at various cut-point values. Github–Local

Figure 11.37 shows the ROC curve for this data.

11.3.5 Multinomial data

When a discrete response variable takes on more than two values, it has a *multinomial* distribution.

nominal: when a response variable can take N distinct values and π_i is the probability of the i^{th} value occurring ($\sum_{i=1}^{N} \pi_i = 1$), then the *baseline-category logit model* (with one explanatory variable, x, in this example) is:

$\log \frac{\pi_n}{\pi_N} = \alpha_n + \beta_i x$, for $n = 1, \ldots, N-1$.

Fitting a model results in $N-1$ equations, with separate coefficients for each.

The `mlogit` package supports the building of multinomial logit models for response variables containing nominal data.

ordinal: fitting an independent logit model to each pair of adjacent values (as is done for nominal models) fails to make use of all the available information; the logit function can be extended to include the ordering information present in ordinal data.

Given an ordinal response variable, Y, that can appear in one of $j = 1, \ldots, N$ possible categories, then Y has a multinomial distribution; its cumulative probability is given by:

$P(Y_i \leq j) = \pi_{i1} + \pi_{i2} + \cdots + \pi_{iN}$, where: π_{ij} is the probability that the i^{th} measurement appears in response category j, and $\sum_{j=1}^{N} \pi_{ij} = 1$

The *cumulative logits* treats $P(Y \leq j)$ as the response variable in a model fitting process that uses a logit link function (other, related functions can be used).

The `ordinal` package supports the building of *cumulative link models*, also known as *ordinal regression models*.

A study by Luthiger and Jungwirth[1163] investigated the importance of fun as a motivation for software development. The survey, which had 1,330 responses from people working on open source projects, asked for an estimate of the percentage of their spare time people spent on activities involving open source development. Possible answers were restricted to intervals of 10%, an ordinal scale. The `clm` functions fits a cumulative link model; `predict` returns a vector of predictions, one for each ordinal value (six vectors in this example):

```
library("ordinal")

f_mod=clm(q42 ~ q31, data=fasd) # Best fitting model is q42 ~ q5+q29+q31

pred=predict(f_mod, newdata=data.frame(q31=1:6))

plot(-1, type="n", xlim=c(1, 6), ylim=c(0, 0.6),
        xlab="Answer given to q31", ylab="Probability\n")

dummy=sapply(1:10, function(X) lines(1:6, pred$fit[ ,X], col=pal_col[X]))
```

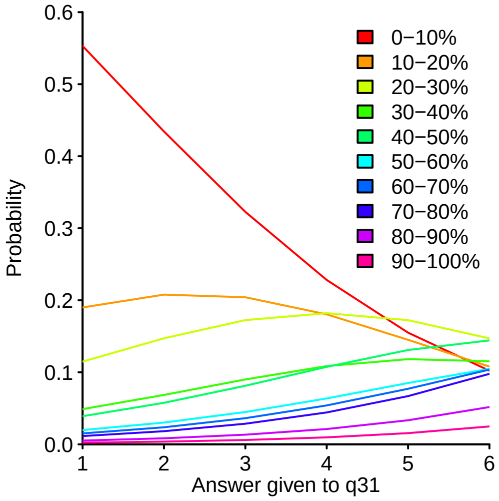


Figure 11.38: Probability of subject response being within a given percentage interval, based on their response to question q31. Data kindly provided by Luthiger[1163] . Github–Local

Figure 11.38 shows the probability of a subject giving an answer within a given 10% band, given their answer to question q31 (the formula: `q42 ~ q5+q29+q31`, is a better fit, but is not easily plotted in 2-D).

11.3.6 Rates and proportions response variables

When dealing with a response variable that is a rate or proportion, there is a fixed lower and upper bound, e.g., 0 and 100. Measurements within a fixed interval often share two characteristics: they exhibit more variation around the mean and less variation towards the lower and upper bounds,[xix], and they have an asymmetrical distribution. These characteristics can be modeled by a Beta equation. A regression model where the response variable is fitted to a Beta equation is known as a *Beta regression model*.

The `betareg` package contains functions that support the fitting of Beta regression models. When fitting basic models, calls to `betareg` have the same form as calls to `glm`; both functions include options that are not supported by the other.

Figure 11.39 shows fitted curves from a beta regression model (red), bootstrapped confidence intervals (blue), and a call to `glm` (green); the study that produced the data is discussed elsewhere, see fig 6.55. The equation fitted is: $mutants_{killed} \propto \sqrt{coverage}$, and was chosen because it is something simple that works reasonably well. Searching for the best fitting exponent, using `nls` (the `betareg` package does not support fitting non-linear models), shows that 0.44 is a better fit than 0.5 for this sample.

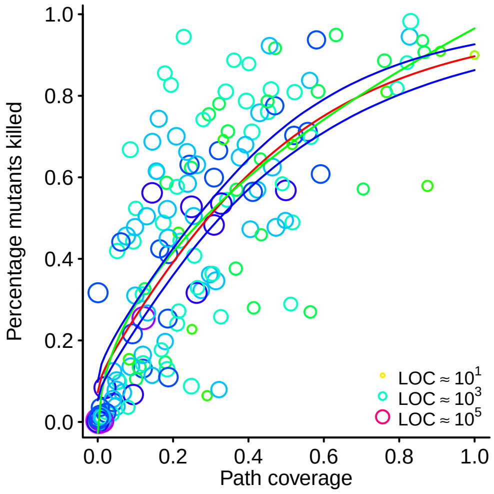


Figure 11.39: Percentage of mutants killed at various percentage of path coverage for 300 or so Java projects; fitted Beta regression (red), with 95% confidence intervals (blue) and `glm` (green) regression models. Data from Gopinath et al.[704] Github–Local

The `summary` output for a Beta regression model includes extra information, as follows: Github–Local

```
Call:
betareg(formula = y_measure ~ I(x_measure^0.5))

Standardized weighted residuals 2:
    Min      1Q  Median      3Q     Max
-2.6881 -0.6403 -0.1279  0.6399  3.1829

Coefficients (mean model with logit link):
                 Estimate Std. Error z value Pr(>|z|)
(Intercept)       -2.5460     0.1891  -13.46   <2e-16 ***
I(x_measure^0.5)   4.7093     0.3502   13.45   <2e-16 ***

Phi coefficients (precision model with identity link):
      Estimate Std. Error z value Pr(>|z|)
(phi)   4.9641     0.5386   9.217   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Type of estimator: ML (maximum likelihood)
Log-likelihood: 80.37 on 3 Df
Pseudo R-squared: 0.6323
Number of iterations: 13 (BFGS) + 1 (Fisher scoring)
```

[xix] The measurement sample is heteroskedastic.

The phi coefficient (the Greek letter ϕ) is the second coefficient of the fitted Beta distribution, $B(\mu, \phi)$.

The predict function returns the expected value of the response variable, $E(y) = \mu$, but does not support a se.fit option (when passed a Beta regression model). The bootstrap can be used to calculate a confidence interval from the predictions made by many models, as follows:

```
library("betareg")
library("boot")

boot_reg=function(data, indices)
{
cov_data=data[indices, ]
b_mod=betareg(mut_cov ~ I(path_cov^0.5), data=cov_data)
# A vector must be returned, i.e., no data frames
return(predict(b_mod, newdata=data.frame(path_cov=x_vals)))
}

cov_boot=boot(pm_info, boot_reg, R = 4999)

ci=apply(cov_boot$t, 2, function(X) quantile(X, c(0.025, 0.975)))
lines(x_vals, ci[1, ], col=pal_col[3])
lines(x_vals, ci[2, ], col=pal_col[3])
```

The default link function used by the betareg function is logit, the same default link used by glm, for the argument family=binomial.

11.4 Multiple explanatory variables

Linear regression can be used to fit models containing more than one explanatory variable; *multiple regression* is the term used for modeling with more than one explanatory variable (the term *bivariate regression* is sometimes applied to the single explanatory variable+response variable case). In theory there is no limit on the number of explanatory variables, but in practice available processing resources, and the need to hold data in storage set an upper bound.

Visualization is much more complicated when there are multiple explanatory variables. Chapter 8 contains examples for visualizing two variables, and the general approach is to break down multiple regression visualization into pairs of variables.

System performance is affected by many factors; figure 11.40 shows SPECint 2006 results for processors running at various frequencies (upper), color coded by memory chip frequency (center) and name of processor family (lower).

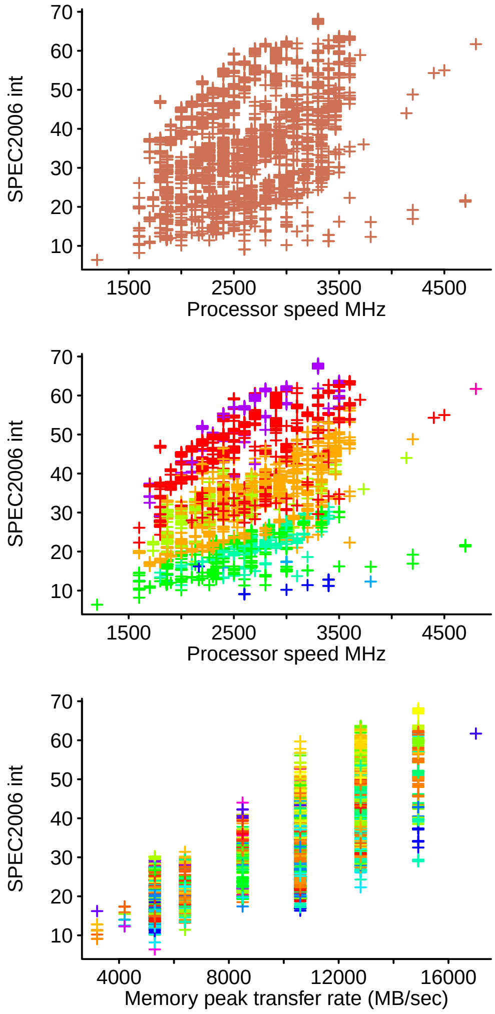


Figure 11.40: SPECint 2006 performance results for processors running at various clock rates, memory chip frequencies and processor family. Data from SPEC.[1720] Github–Local

The SPECint results include 36 columns of information relating to the benchmarked system. Which of these columns contains information that can be used to succinctly model the performance of a system, and what equation best describes the form of their contribution?

The R formula notation includes a symbol that denotes all columns in the data frame as explanatory variables, except the one specified as the response variable; the dot symbol is used as follows:

```
spec_mod=glm(Result ~ ., data=cint)
```

Given enough cpu power and memory, it can be more productive to start by considering all explanatory variables and remove underperforming variables, rather than starting with the explanatory variables believed to be the most important and then adding more variables.

The stepAIC function, in the MASS package, automates the process of removing underperforming explanatory variables from a fitted model, to create a model having a minimum AIC (the step function, in the base system, is a rather minimal implementation).[xx]

When some domain knowledge is available (e.g., performance often correlates with clock rate and is not usually affected by date of execution), experimentation of fitting models

[xx] This fishing expedition approach to model building requires that p-values be suitably reduced, e.g., using a Bonferroni corrected value.

containing explanatory variables considered to be most likely to have a large impact on the response variable, can help refine the analyst's appreciation of the impact of different explanatory variables on overall performance.

For this SPEC dataset, there is so much detail recorded in the `Processor` column of the Spec results, that each entry is often unique; making it possible to create an almost perfect, but completely uninformative, model using just this one explanatory variable.

The following model explains 80% of the variance in the `Result` values:

```
spec_mod=glm(Result ~ Processor.MHz+mem_rate+mem_freq, data=cint)
```

where: `Processor.MHz` is the processor clock rate, `mem_rate` the peak memory transfer rate and `mem_freq` the frequency at which memory is clocked.

The + binary operator, in the above formula, specifies that explanatory variables are added together. The `summary` output shows that the equation fitted by `glm` is:

$$Result = -2.4\cdot 10^{1} + Processor.MHz 7.3\cdot 10^{-3} + mem\_rate 2.5\cdot 10^{-3} + mem\_freq 1.0\cdot 10^{-2}$$

With a single explanatory variable, it is easy to visually compare model predictions against measured values; with multiple variables things are not so simple. One approach is to analyse the impact of each variable, on predictions, in turn.

The `crPlot` and `crPlots` functions, in the `car` package, produce a *component+residual* plot (also known as a *partial-residual* plot); the y-axis contains the predicted value plus the residual, the x-axis contains the value of the explanatory variable:

```
library("car")

spec_mod=glm(Result ~ Processor.MHz+mem_rate+mem_freq, data=cint)

crPlots(spec_mod, term= ~ ., layout=c(3, 1), col=point_col,
          cex.lab=1.5, cex.axis=1.5, ylab="Component+Residuals\n", main="")
```

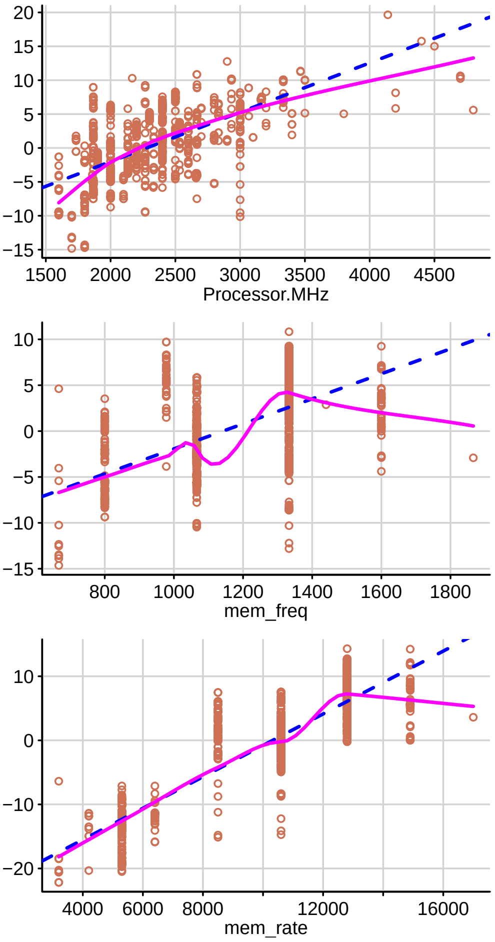


Figure 11.41: Component+residual plots for three explanatory variables in a fitted SPECint model. Github–Local

Figure 11.41 shows the component+residual plots produced by the above code. The red dotted line is derived from the fitted model, and the green line a loess fit; if the form of an explanatory variable, in the formula used to fit a model, is close to reality, the two lines will be closely intertwined. For the SPEC model there is consistent divergence of the two lines, over ranges of the measurement interval, for two variables and perhaps some for a third.

Experience of hardware characteristics suggests that performance does not increase forever, as clock rates are increased. Adding quadratic forms of the explanatory variables to the model is a step up in complexity, to try out with a fitted model (an exponential is more realistic, in that its maximum converges to a limit, but this form of modeling requires the use of non-linear regression, which is covered later).

Adding quadratic terms, for two of the three explanatory variables, to the fitted model explains another 4% of the variance, but significantly reduces the error at higher processor and memory frequencies; see figure 11.42.

Some of the systems benchmarked contained error correcting memory, which might be expected to slightly reduce performance. The `update` function can be used to add, or remove, explanatory variables from a previously fitted model. The following code adds the variable `ecc` to the previously fitted model, `spec_mod`:

```
ecc_spec_mod=update(spec_mod, . ~ . + ecc)
```

The advantage of using `update` is a reduction in the system resources needed to fit the model, compared with starting from the beginning again.

The `summary` output shows that systems containing error correcting memory have slightly better performance. Before jumping to the conclusion that adding error correction improves system performance, it is worth noting that this kind of memory tends to be used in high-end systems, where it is likely that money has been spent to improve performance and reliability.

The choice of cpu and memory frequency is based on information that is not present in the SPEC result data, the intended price point the computing system is designed to be sold at, and the trade-off in the cost/performance of the components needed to build it.

What contribution does each explanatory variable make to a fitted model? Some ways in which individual contributions can be measured include:

- the amount of variance, in the response variable, explained by an explanatory variable.

  The `calc.relimp` function, in the `relaimpo` package, calculates the contribution made by each explanatory variable to the variance explained by the fitted model,

- the impact each explanatory variable can have on the range of values taken by the response variable (with all other explanatory variables maintaining a fixed value).

  For very simple models, [xxi] one way of calculating the maximum impact on the value of the response variable is by multiplying the minimum/maximum value taken by an explanatory variable by the corresponding coefficient in the fitted model. For instance, `range(cint$Processor.MHz)*7.3*10^-3` evaluates to `11.68 35.04`, a difference of `23.36`.

  The `visreg` function, in the `visreg` package, produces a visual representation of the impact of each explanatory variable on the response variable.

  Nomograms are a visual method for calculating the value of a response variable when each explanatory variable has a particular value: the `DynNom` function in the `DynNom` package supports interactive exploration of model behavior in a web browser.

  Normalising values prior to fitting a model is sometimes suggested (e.g., using the `scale` function); the relative values of the model coefficients can then be directly compared. This method only works when all explanatory variable values are drawn from a Normal distribution.

The `relaimpo` package supports a variety of functions[737] for calculating the relative contribution made to a model by each explanatory variable it contains. For instance, the `calc.relimp` function calculates: `first`, the variance explained by a model containing just each variable, `last`, the variance explained when a variable is added to a model that already contains the other variables, `betasq`, the standardized coefficients of the model (i.e., one fitted after normalising the data; effectively a metric for the contribution of each explanatory variable to the response variable value), and `lmg` (named after the initials of its creators), the variance explained by each variable; the `boot.relimp` function returns confidence intervals for these values.

```
library("relaimpo")

spec_mod=glm(Result ~ Processor.MHz+I(Processor.MHz^2)+mem_rate
                              + I(mem_rate^2)+mem_freq, data=cint)

# How much does each explanatory variable contribute?
calc.relimp(spec_mod, type = c("first", "last", "betasq", "lmg"))
```

The `calc.relimp` output is: Github–Local

```
Response variable: Result
Total response variance: 81.56
Analysis based on 1346 observations

5 Regressors:
Processor.MHz I(Processor.MHz^2) mem_rate I(mem_rate^2) mem_freq
Proportion of variance explained by model: 83.77%
Metrics are not normalized (rela=FALSE).

Relative importance metrics:

                       lmg     last   first betasq
Processor.MHz      0.06189 0.017807 0.04609 0.6888
I(Processor.MHz^2) 0.04556 0.005698 0.02962 0.2201
mem_rate           0.29380 0.028909 0.55554 2.0853
I(mem_rate^2)      0.29050 0.006363 0.58253 0.4768
mem_freq           0.14598 0.067531 0.28997 0.1258

Average coefficients for different model sizes:

                          1X        2Xs        3Xs        4Xs        5Xs
Processor.MHz      4.308e-03  1.290e-02  1.568e-02  1.823e-02  1.666e-02
I(Processor.MHz^2) 6.560e-07 -4.894e-07 -9.880e-07 -1.728e-06 -1.788e-06
mem_rate           2.343e-03  1.562e-03  2.236e-03  3.303e-03  4.540e-03
I(mem_rate^2)      1.280e-07  1.488e-07  8.108e-08 -1.286e-08 -1.158e-07
mem_freq           2.429e-02  2.012e-02  1.558e-02  1.457e-02  1.600e-02
```

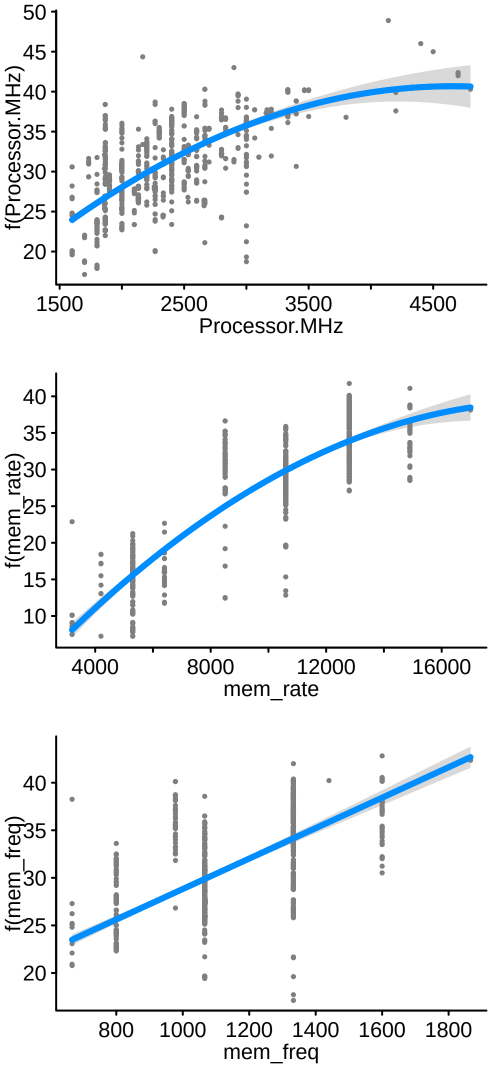


Figure 11.42: Individual contribution of each explanatory variable to the response variable in a quadratic model of SPECint performance. Github–Local

[xxi] Those that are linear in the explanatory variable, with no interactions between variables.

the second set of columns, under the line starting `Average coefficients`, lists the model coefficients for each explanatory variable, if that variable were to appear in a model containing X variables (values are averaged over all combinations of other variables). The values in the last column (5Xs in this case) are the same as those produced by the `summary` function for the fitted model.

How do changes in the value of each explanatory variable individually affect the value of the response variable? The `visreg` package supports functions for plotting the relationship between the response variable and individual explanatory variables (with the other variables held constant at their median value).

Figure 11.42 shows the individual contribution made by each explanatory variable to the value of the response variable (along with confidence intervals in grey), for the following model of SPECint performance:

```
library("visreg")

spec_mod=glm(Result ~ Processor.MHz + I(Processor.MHz^2)+mem_freq
                                   +mem_rate+I(mem_rate^2), data=cint)
visreg(spec_mod)
```

Figure 11.43 shows a contour plot created using the `visreg2d` function.

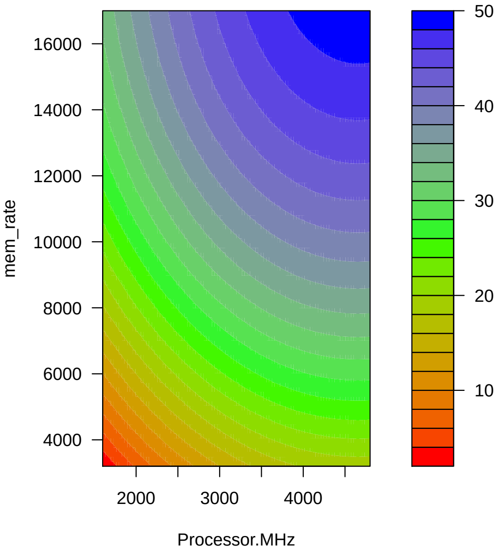


Figure 11.43: Contour map of `Result` values predicted by a fitted model of SPECint performance, over range of `Processor.MHz` and `mem_rate` values. Github–Local

Sometimes including an explanatory that has no correlation with the response variable improves the performance of a model; why does this happen? An explanatory variable may correlate with the residual of a model, and adding this new variable has the effect of improving a model by reducing its residual.

11.4.1 Interaction between variables

In the models fitted so far, each explanatory variable has been independent of the others. The `glm` function and many other regression modeling functions provide mechanisms for specifying interactions between explanatory variables, using binary operators in the formula, such as `:`, `*` and `^`.

Operator	Effect
+	causes both of its operands to be included in the equation.
:	denotes an interaction between its operands, e.g., `a:b` or `a:b:c`.
*****	denotes all possible combinations of + and : operators, e.g., `a*b` is equivalent to `a+b+a:b`.
^	denotes all interactions to a specific degree, e.g., `(a+b+c)^2` is equivalent to `a+b+c+a:b+a:c+b:c`.
.	denotes all variables in the data-frame specified in the `data` argument except the response variable.
-	specifies that the right operand is removed from the equation, e.g., `a*b-a` is equivalent to `b+a:b`.
-1	specifies that an intercept is not to be fitted (many regression fitting functions implicitly include an intercept).
I()	"as-is", any operators in the enclosed expression are not treated as formula operators, the behavior is that applying outside a formula.

Table 11.2: Symbols that can be used within a formula to express relationships between explanatory variables.

As with all data analysis, the choice of interactions between explanatory variables should be driven by domain knowledge. When there is a lot of uncertainty about which interactions are significant, it may be easiest to start by specifying all pairs of interactions between variables (or triple interactions if there are not too many variables), and to then simplify, either automatically using `stepAIC`, or through manual inspection of `summary` output of the fitted models.

Stepwise regression techniques, such as that provided by `stepAIC`, can return models that suffer from a variety of problems, such as overfitting. There are techniques available to help avoid these problems; the `train` function in the `caret` package supports some of these techniques. The `glmulti` package automates the process of finding an optimal, in a sense specified by the user (e.g., minimise AIC or some other measure), explanatory variable interaction; a list of variables is specified, and the function permutes through the possibilities, e.g., `glmulti("y", c("a", "b", "c", "d"), data=some_data)`.

A study by Moløkken-Østvold and Furulund[1289] investigated the impact of daily communication between the customer and contractor on the accuracy of effort estimates, for 18 software projects. Figure 11.44 shows estimated vs. actual effort broken down by communication frequency (i.e., daily or not daily), along with individually fitted straight lines.

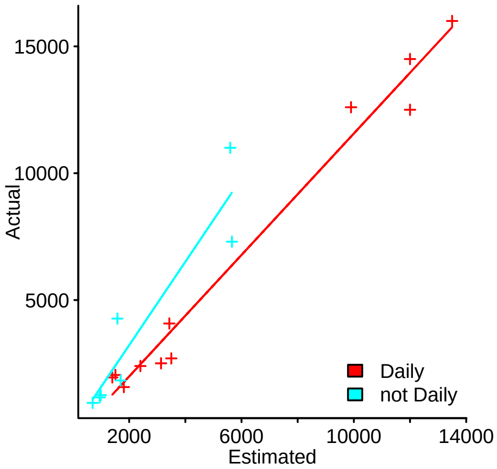


Figure 11.44: Estimated and actual effort broken down by communication frequency, along with individually fitted straight lines. Data from Moløkken-Østvold et al.[1289] Github–Local

It is possible to fit one regression model that simultaneously fits both straight lines to this data; the following code shows one possibility:

```
sim_mod=glm(Actual ~ Estimated+Estimated:Communication, data=sim)
```

The fitted equation is (based on summary output):

$$\begin{aligned} Actual &= -270.1 + 1.18 Estimated + 0.51 Estimated \times D \\ &= -270.1 + (1.18 + 0.51 D) Estimated \end{aligned}$$

where: *Actual* is the actual and *Estimated* the estimated effort, and D has one of two values:

$$D = \begin{cases} 1 & \text{daily communication} \\ 0 & \text{not daily communication} \end{cases}$$

Is this formula the best fit possible using the available data? The formula used was selected by your author, because of a belief that the benefit of communication will increase as project size increases.

There are six data points for each of the 18 projects, computationally small enough for the brute force approach of examining all possible models; but with only 18 projects, some formula possibilities cannot be fitted because they contain more variables than available data points (a unique solution requires fewer variables than data points).

The formula in the following code fits four explanatory variables individually, plus each variable paired with every other variable (one at a time). stepAIC is used as a quick way of removing explanatory variables that are not paying their way (automatic model selection is fraught with problems, with perhaps the largest being that it allows analysts to stop thinking about the data):

```
sim_mod=glm(Actual ~ (Estimated+Communication+Contract+Complexity)^2, data=sim)

min_sim=stepAIC(sim_mod)
summary(min_sim)
```

This book fits regression models as a means of building understanding, and minimising AIC is often a useful step along the way. Another way of removing low impact variables from a model is to consider the p-value of each fitted component.

The summary output for ordinal and nominal explanatory variables, lists p-values for each value that these variables take in the data. The Anova function (in the car package) lists p-values at the variable level, and its output for the above model is: Github–Local

```
Analysis of Deviance Table (Type II tests)

Response: Actual
                       LR Chisq Df Pr(>Chisq)
Estimated                45.879  1  1.258e-11 ***
Communication            17.272  1  3.240e-05 ***
Contract                  6.767  3    0.07971 .
Complexity                1.543  1    0.21423
Estimated:Communication   2.546  1    0.11060
Communication:Contract    5.020  3    0.17034
Contract:Complexity       3.197  2    0.20224
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

The variable Complexity has the highest p-value, and repeatedly removing the component having the highest p-value, for successively smaller models (smaller in the sense of containing fewer components) leads to the following model:

```
sim_mod=glm(Actual ~ Estimated+Communication+Communication:Contract, data=sim)
```

which fits the following equation:

$$\begin{aligned} Actual = -274.8 + 1.21 Estimated + 2625 \times !D + \\ C_{fp}(1862 \times !D - 197.6 \times D) + \\ C_{tp}(-2270 \times !D - 462.2 \times D) + \\ C_{ot}(-2298 \times !D - 234.3 \times D) \end{aligned} \qquad (11.6)$$

where the new variables are: C_{fp} is a fixed price contract, C_{tp} is a target price contract and C_{ot} other kind of contract.

$$C_{fp} = \begin{cases} 1 & \text{fixed price contract} \\ 0 & \text{not fixed price contract} \end{cases} \quad C_{tp} = \begin{cases} 1 & \text{target price contract} \\ 0 & \text{not target price contract} \end{cases}$$

$$C_{ot} = \begin{cases} 1 & \text{other contract} \\ 0 & \text{not other contract} \end{cases}$$

This model explains a greater percentage of the variance in the data than the first model fitted, it also has a slightly smaller AIC. While it makes use of extra information (i.e., the kind of contract), a more noticeable difference is that `Communication` has a constant effect (i.e., it does not increase with estimated size); the case of fixed price contracts, with no daily communication cries out for attention.

Following the numbers has produced a model that is a better fit to the data, but not to expectations (which may, of course, be wrong).

11.4.2 Correlated explanatory variables

The mathematics behind many approaches used to fit linear regression models assumes that explanatory variables are independent of each other. If a linear relationship exists between one or more pairs of explanatory variables (i.e., a relationship of the form: $PV_1 = a + b \times PV_2$, where PV_1 and PV_2 are explanatory variables, a is any constant and b a non-zero constant), then this needs to be taken into account by the model building technique used.[xxii]

Multicolinearity is said to occur, when a linear relationship exists between two or more explanatory variables, the term *colinearity* is often used when only two variables are involved.

Figure 11.45 illustrates how the variance in Y explained by combining X_1 and X_2 may be less than the sum of the variance explained by each individually, because the two variables are not independent; there is a shared contribution.

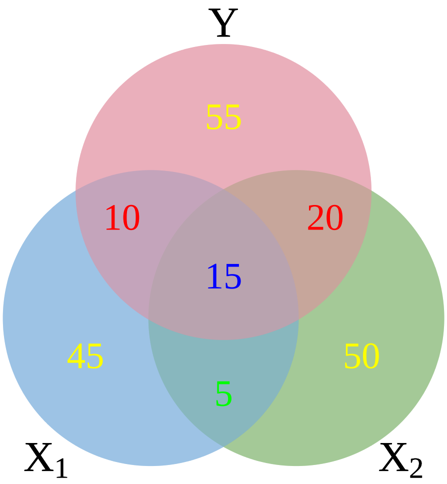


Figure 11.45: Illustration of the shared and non-shared contributions made by two explanatory variables to the response variable Y. Github–Local

The impact of multicolinearity is to increase the standard error in the calculated value of the fitted model coefficients (i.e., the β_n), potentially resulting in a model that is not considered acceptable or is unreliable (in the sense that small changes in the data result in large changes in the coefficients of the fitted model). The increased uncertainty, in some variables, will make it more difficult to isolate the effects of individual explanatory variables and will increase the width of the confidence intervals for the predicted values of the response variable.

The *Variance Inflation Factor* (VIF) is a measure of the uncertainty created by the presence of multicolinearity. The impact of VIF is the same as reducing the sample size. When no multicolinearity is present, VIF has a value of one. The impact on the standard error is:

$$\varepsilon_{standard} \propto \sqrt{\frac{VIF}{observations}}$$

When is a VIF value too large? A large VIF is more likely to be acceptable with a large sample, compared to a small one, e.g., the standard error is proportionally the same for 10,000 observations having a VIF of 400 and for 100 observations having a VIF of 4.

Suggested maximum VIF values appear in print, e.g., 5 or 10 are sometimes suggested. As always, think about what the VIF value means in the context of how the results will be used; pick a value that makes sense given the sample size, the error in the measurements and the level of error that is acceptable in the business context.

[xxii] It is ok for a nonlinear relationship to exist in linear models, e.g., $PV_1 = a + b \times PV_2^2$.

The car and rms packages support a vif function, that takes the model returned by a call to, for instance, glm and returns the VIF for each explanatory variable.

A study by Kroah-Hartman[1033] investigated the amount of change in the Linux kernel source code occurring between each release. Figure 11.46 shows the number of lines added, modified and removed, plus overall growth, number of files and total number of lines at each initial release of the Linux kernel from version 2.6.0 to 3.9 (two outliers have been excluded).

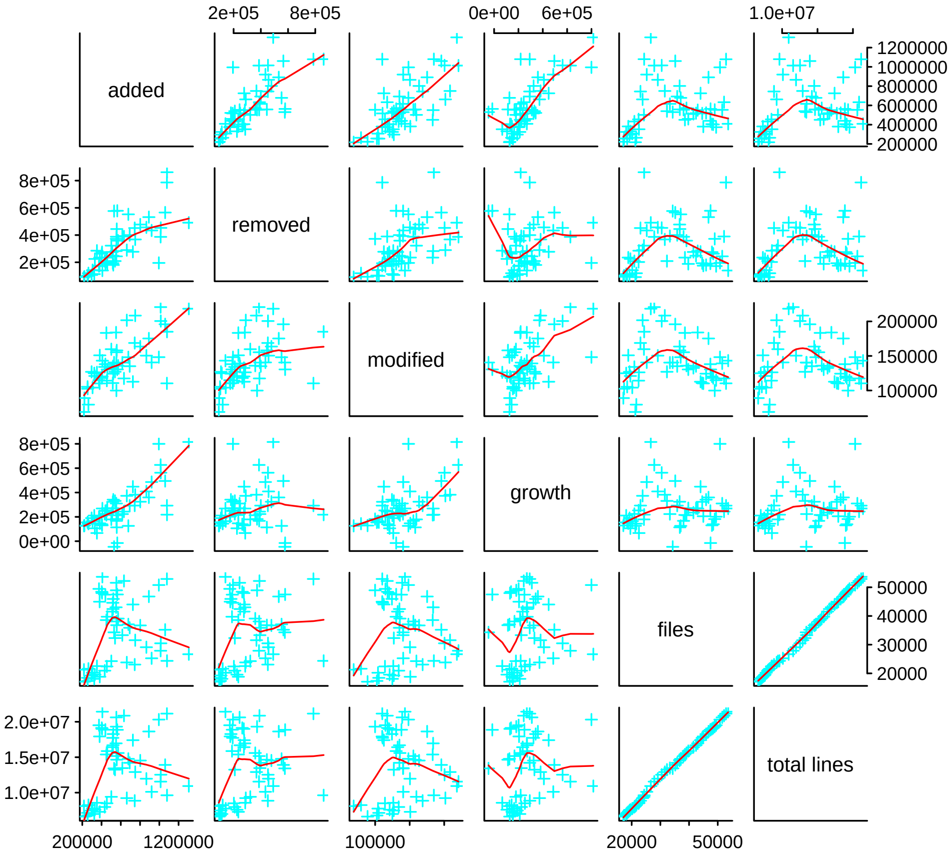


Figure 11.46: pairs plot of lines added/modified/removed, growth and number of files and total lines in versions 2.6.0 through 3.9 of the Linux kernel. Data from Kroah-Hartman.[1033] Github–Local

Building a model of the Linux kernel growth is complicated by the potentially high correlation between some measured variables, including:

- the growth, in lines of code, between releases is the difference between lines added and lines removed; these three variables are perfectly correlated in that knowing two of them enables the third to be calculated,
- lines added appears strongly correlated with lines removed. Perhaps existing functionality is being rewritten, rather than unrelated functionality being added,
- the decision about whether a line has been modified or removed/added is made algorithmically (rather than asking the developer who made the change). The amount of misclassified lines is not known,
- system level measurements are also correlated, e.g., number of files and total lines of code.

Modeling the number of modified lines, using the Kroah-Hartman data, finds that both lines added and lines removed individually explain around half of the deviance (61% and 41% respectively). However, combining them in a model does not produce any improvement; the following output from summary was obtained by including the argument correlation=TRUE. Github–Local

```
Call:
glm(formula = lines.modified ~ lines.added + lines.removed, data = amr_out)

Deviance Residuals:
   Min      1Q  Median      3Q     Max
-72376  -12049     321   11274   54964

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept)    8.705e+04  8.625e+03  10.093 9.40e-14 ***
lines.added    9.958e-02  2.093e-02   4.759 1.64e-05 ***
lines.removed -1.500e-02  3.117e-02  -0.481    0.632
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 639477748)

    Null deviance: 6.2276e+10  on 53  degrees of freedom
Residual deviance: 3.2613e+10  on 51  degrees of freedom
AIC: 1253.1

Number of Fisher Scoring iterations: 2

Correlation of Coefficients:
              (Intercept) lines.added
lines.added   -0.54
lines.removed -0.06       -0.76
```

The correlation between the model coefficients appears at the end of the output and shows a high negative correlation between `lines.added` and `lines.removed`; the variable `lines.added` is a better predictor of `lines.modified` and has been selected over `lines.removed` (whose p-value is significantly larger than when just this variable appeared in a model).

A call to the `vif` produces the following: Github–Local

```
  lines.added lines.removed
      2.39311       2.39311
```

With only two explanatory variables, there is no ambiguity about which variables are involved in a linear relationship, but with more than two variables things are not always so obvious. The correlation table produced by `summary` can be used to identify related variables; the `alias` function generates just this information, when the argument `partial=TRUE` is specified.

Approaches to dealing with multicolinearity, to reduce any undesirable impact it may have on fitting a model, include:

- removing one or more of the correlated explanatory variables. The choice of which explanatory variables to remove might be driven by:
  - the cost of collecting information on the variable(s),
  - a VIF driven approach. The process involves fitting a model using the current set of explanatory variables, removing the explanatory variable with the largest VIF (removing one variable affects the VIF of those that remain and may reduce the VIF of other variables to an acceptable level) and iterating until all explanatory variables have what is considered to be an acceptable VIF,
- combining the strongly correlated variables in a way that makes use of all the information they contain.

The disadvantages of excluding explanatory variables from a model include:

- ignoring potentially useful information present in the excluded variable,
- creating a model that gives a false impression about which explanatory variables are important, i.e., readers will assume that the variables appearing in the model are the only important ones, unless information about the excluded variables is also provided,
- it provides the opportunity for the analyst to select the model that favours the hypothesis they want to promote (by selecting which explanatory variables appear in the model).

The SPEC power benchmark[1719] is designed to measure single and multi-node server power consumption, while executing a known load. The results contain 515 measurements of six system hardware characteristics, such as number of chips, number of cores and total memory, as well as average power consumption at various load factors.

A model of average power consumption, at 100% load, containing a linear combination of all explanatory variables, shows very high multicolinearity for the number of chips (its VIF is 27.5 and several other variables have a high VIF; see Github–hardware/SPECpower.R).

Removing this variable reduces the VIF of the remaining variables, but the AIC drops from 6798.7 to 7182.1. Whether this decrease in model performance is important depends on the reason for building the model, e.g., prediction or understanding. Do the values of the model coefficients, after removing this variable, provide more insight than the coefficient values of the original model? These kinds of questions can only be answered by a person having detailed domain knowledge. This example shows how removing a variable solves one problem and raises others.

A study of fault prediction by Nagappan, Zeller, Zimmermann, Herzig and Murphy[1327] produced data containing six explanatory variables having an exact linear relationship with other explanatory variables. The `glm` function detects the existence of this relationship and makes a decision about explanatory variables to exclude from the model (the value returned for their fitted coefficients is `NA`); see Github–regression/change-burst-sum.R.

Two explanatory variables having an exact linear relationship will have a correlation of ±1, as a call to `alias` will show.

11.4.3 Penalized regression

Penalized regression handles multicolinearity by automatically selecting how much each explanatory variable should contribute to the model; explanatory variables are penalized, based on their relative contribution to the model. The `penalized` package supports penalized regression.

The traditional technique for fitting a regression model involves minimising some measure of a specified error, where the error is defined to be the difference between actual and predicted values, e.g., the sum of squared error, whose equation is:

$$SSE = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Penalised regression modifies this equation to include a penalty (the λ in the equation below), for the P coefficients in the model (β in the equation below).

$$SSE_{enet} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \lambda_1 \sum_{j=1}^{P} |\beta_j| + \lambda_2 \sum_{j=1}^{P} \beta_j^2$$

This technique of using both first- and second-order penalties is known as the *elastic net*. When only the first order term, λ_1, is used, it is known as the Least Absolute Shrinkage and Selection Operator method (*lasso*). When only the second order term, λ_2, is used, it is known as *ridge regression*.

In theory the penalization penalties, λ_1 and λ_2, are chosen by the analyst. In practice, software packages provide a function that automatically finds values (using the bootstrap) that minimise the error.

The lasso tends to pick one from each set of correlated variables and ignores the rest (by setting the corresponding βs to zero). Ridge regression has the effect of causing the coefficients, β, of the corresponding correlated variables to converge to a common value, i.e., the coefficient chosen for k perfectly correlated variables is $\frac{1}{k}$th the size chosen, had just one of them been used.

The calculation of mean squared error (MSE) adds contributions from both variance and bias. The default regression modeling techniques are unbiased (i.e., they attempt to minimise bias). It is possible to build models with lower MSE by trading off bias for variance (see Github–hardware/SPECpower.R for an example; in this case the use of penalised regression makes little difference to the final model).

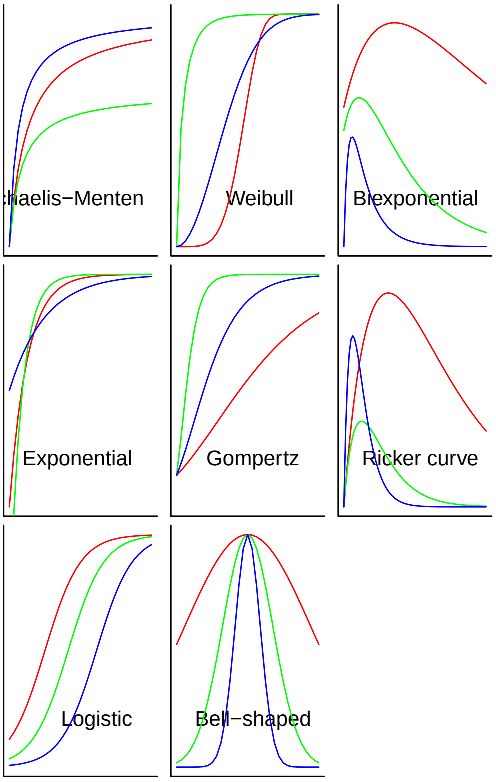


Figure 11.47: Example plots of functions listed in table 11.3. These equations can be inverted, so they start high and go down. Github–Local

11.5 Non-linear regression

The regression models fitted in earlier sections are linear models because the coefficients of the model (e.g., β_1 in the equation at the start of this chapter) are linear (the form of the explanatory variables is irrelevant). In a non-linear regression model one or more of the coefficients have a non-linear form, e.g., θ_1 in the following equation:

$$y = \alpha_1 + \beta_1 x^{\theta_1} + \varepsilon$$

Table 11.3 lists some commonly occurring non-linear equations, and figure 11.47 illustrates example instances of these equations.

The `nls` function (Nonlinear Least Squares) is part of the base system and can be used to build non-linear regression models; it requires that the response variable error have a Normal distribution (`glm`'s default behavior). The `gnm` package (Generalized nonlinear models) contains support for other forms of error distribution.

Shape	**Name**	**Equation**
Asymptotic growth to a limit	Michaelis-Menten	$y = \frac{ax}{1+bx}$
Asymptotic growth to a limit	Exponential	$y = a(1-be^{-bx})$
S-Shaped	Logistic	$y = a + \frac{b-a}{1+e^{(c-x)/d}}$
S-Shaped	Weibull	$y = a - be^{-cx^d}$
S-Shaped	Gompertz	$y = ae^{be^{-cx}}$
Humped	Bell-shaped	$y = ae^{-\lvert bx \rvert^2}$
Humped	Biexponential	$y = ae^{-bx} - ce^{-dx}$
Humped	Ricker curve	$y = axe^{-bx}$

Table 11.3: Some commonly encountered non-linear equations, see figure 11.47.

From the practical point of view there are several big differences between using `glm` and using `nls`, including:

- `nls` may fail to fit a model; the techniques used to find the coefficients of a non-linear model are not guaranteed to converge,
- `nls` may return a fitted model that differs from the actual solution; the techniques used to find the coefficients of a non-linear model may become stuck in a local minimum, that is good enough, and fail to find a better solution,
- `nls` often requires the analyst to provide estimate(s) for the initial value of each model coefficient, that is close to the final values (using the `start` argument),
- names for the model coefficients being estimated have to explicitly appear in the formula (i.e., implicit names are not created automatically),
- the operators appearing in the expression to the right of ~ have their usual arithmetic interpretation, i.e., the formula specific behaviors listed in table 11.2 do not apply.

The biggest problem with fitting non-linear regression models, is finding a combination of starting values that are good enough for `nls` be able to converge to a fitted model. Possible techniques for finding these values include:

- using a "self-start" function, if available (e.g., `SSlogis` for Logistic models); these attempt to find good starting values to feed into `nls`, and functions, in turn, may require starting values (but at least there is a known method for calculating them),
- fitting a linear model that is close enough to the non-linear model and working with the coefficients of the fitted linear model as possible starting values,
- using the argument `trace=TRUE`, which outputs the list of model coefficients that are being used internally, as a source of ideas,
- picking a few points in the plotted data that a fitted line is likely to pass through and calculating values that would result in the equation being fitted, passing close to these points.

A study by Hazelhurst[787] measured the performance of various systems running a computational biology program. Figure 11.48 shows four non-linear equations fitted to one processor characteristic (L2 cache size). The calls to `nls` are as follows:[xxiii]

```
b_mod=nls(T1 ~ c+a*exp(b*L2), data=bench, start=list(a=300, b=-0.1, c=60))

mm_mod=nls(T1 ~ (1+b*L2)/(a*L2), data=bench, start=list(b=3, a=0.004))

gm_mod=nls(T1 ~ a/exp(b*exp(-c*L2)), data=bench,
                start=list(a=80, b=-1, c=0.1), trace=FALSE)
Asym = 0.0125
Drop = 0.002
lrc = -1.0
```

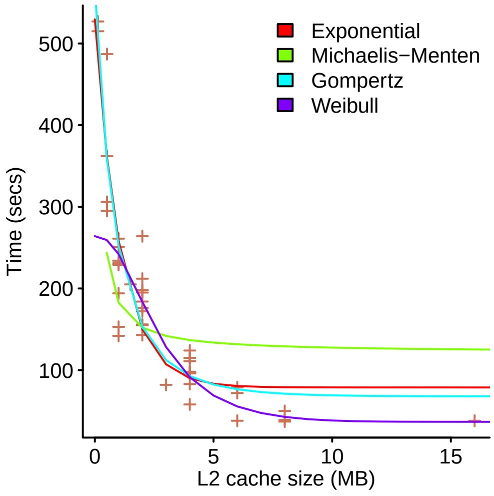


Figure 11.48: Time to execute a computational biology program on systems containing processors with various L2 cache sizes. Data kindly provided by Hazelhurst.[787] Github–Local

[xxiii] It is difficult to separate inspiration from suck it and see, in this process.

```
pwr = 2.5
# 1/SSweibull does not have the desired effect, so have to invert the response.
getInitial(1/T1 ~ SSweibull(L2, Asym, Drop, lrc, pwr), data=bench)
wb_mod=nls(1/T1 ~ SSweibull(L2, Asym, Drop, lrc, pwr), data=bench)
```

At the start of this chapter, various linear models were fitted to the growth of Linux, see fig 11.7. Polynomials containing integer powers were used, perhaps the data is better fitted by a polynomial containing non-integer powers. The following call to nls attempts to fit such an equation, it uses starting values extracted from the quadratic model fitted earlier:

```
m1=nls(LOC ~ a+b*Number_days+Number_days^c, data=h2,
                        start=list(a=3e+05, b=-4e+2, c=2.0))
```

The summary and AIC output is: Github–Local

```
Formula: LOC ~ a + b * Number_days + Number_days^c

Parameters:
    Estimate Std. Error t value Pr(>|t|)
a -1.679e+05  2.969e+04  -5.656 2.61e-08 ***
b  7.319e+02  3.463e+01  21.131  < 2e-16 ***
c  1.806e+00  4.616e-03 391.211  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 231800 on 498 degrees of freedom

Number of iterations to convergence: 5
Achieved convergence tolerance: 4.299e-06

[1] "AIC =  13805.2100165816"
```

showing that the equation:

$$sloc = (-1.68\cdot 10^5 \pm 3\cdot 10^4) + (7.32\cdot 10^2 \pm 3.5\cdot 10^1)Number\_days + Number\_days^{1.81\pm 4.6\cdot 10^{-3}}$$

is a slightly better fit than a cubic equation (i.e., a lower AIC) and also predicts continuing growth (unlike the cubic equation).

It is possible that further experimentation will find a polynomial model with a lower AIC. However, the purpose of this analysis is to understand what is going on, not to find the equation whose fitted model has the lowest AIC.

A more practical issue is to create a model that makes what are considered to be more realistic future predictions. The growth in the number of lines in the Linux kernel will not continue forever, at some point the number of lines added will closely match the number of lines deleted. One commonly seen growth pattern, starts slow, has a rapid growth period, followed by a levelling off converging to an upper limit (i.e., an S-shaped curve). The Logistic equation is S-shaped and is often used to model this pattern of growth; the equation involves four unknowns (third row in table 11.3).

Fitting a Logistic equation the hard and easy (when it works) way:

```
# suck it and see...
m3=nls(LOC ~ a+(b-a)/(1+exp((c-Number_days)/d)), data=h2,
                 start=list(a=-3e+05, b=4e+6, c=2000, d=800))
# no thinking needed, SSfpl works out of the box for this data :-)
m3=nls(LOC ~ SSfpl(Number_days, a, b, c, d), data=h2)
```

The AIC for the fitted Logistic equation is slightly worse than the cubic polynomial (13,273 vs. 13,220), but a lot better than the quadratic fit and it predicts a future trend that is likely to occur, eventually.

While the predict function includes parameters to request confidence interval and standard error information, support for both is currently unimplemented for models fitted using nls. The confint function in the MASS package, when passed a model built using nls, returns the confidence intervals for each model coefficient; bootstrapping can also be used to find confidence intervals.

Figure 11.49 shows the fitted model predicting a slow down in growth, with the maximum being reached at around 10,000 days. Who is to say whether this prediction is more likely

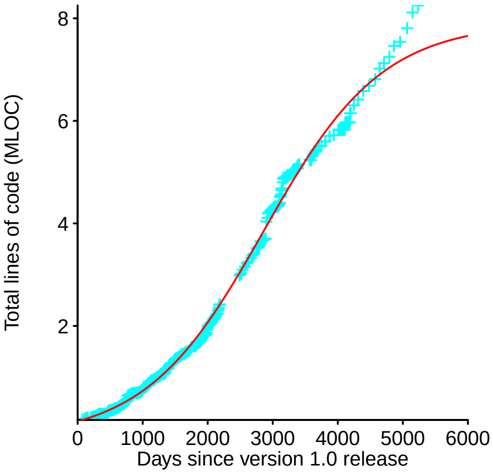


Figure 11.49: A logistic equation fitted to the lines of code in every non-bugfix release of the Linux kernel since version 1.0. Data from Israel et al.[891] Github–Local

to occur, over the specified number of days, than the continuing increase predicted by the quadratic model? Given that the one explanatory variable used to fit the models, time, does not directly impact the production of source, it is no surprise that the predictions of future behavior made by the various models vary so wildly.

One technique for getting a rough idea of the accuracy of the future predictions made by a model, is to fit models to subranges of the data, and then check the predictions made against the known data outside the subrange. Figure 11.50 shows logistic equations fitted to subranges of the data, e.g., all data up to 2900, 3650, 4200 number of days and all days.

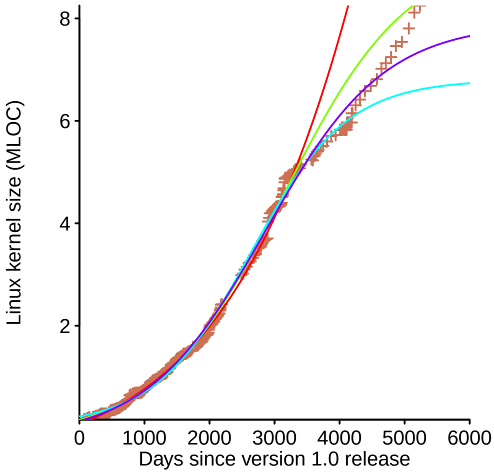


Figure 11.50: Predictions by logistic equations fitted to Linux SLOC data, using subsets of data up to 2900, 3650, 4200 number of days and all days since the release of version 1.0. Data from Israel et al.[891] Github–Local

The lesson to learn from figure 11.50 is to be careful what you ask for, asking for a logistic equation fitted to the data may get you one. The fitting process is driven by your expectations (in the form of a formula), and the data it is given.

The processes generating the data fitted by a Logistic equation may not in themselves follow this pattern, the contributions of independent processes may combine to create an emergent pattern. A study by Grochowski and Fontana[736] showed that increases in the density of data stored on hard disks could be viewed as a sequence of technologies that each rapidly improved (e.g., magneto-resistive and antiferromagnetically-coupled). Figure 11.51 shows the areal density (think magnetic domains) of various models of hard disk on first entering production. Improvements in each technology can be fitted with its own Logistic equation, as can the overall pattern of performance improvements.

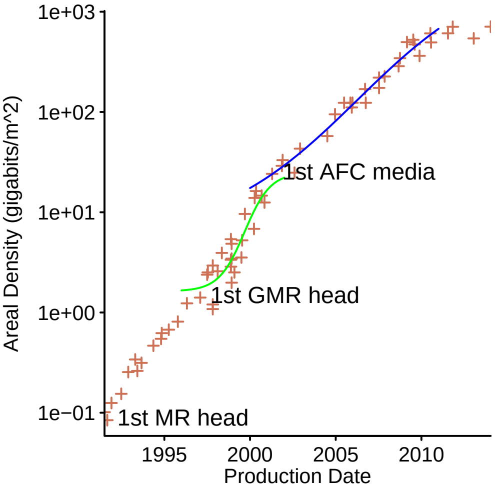


Figure 11.51: Increase in areal density of hard disks entering production over time. Data from Grochowski et al.[736] Github–Local

A codebase showing some evidence of having completed its major expansion phase is glibc, the GNU C library (i.e., its growth rate has levelled off); see figure 11.52. The `summary` of the fitted model is (the `SSfpl` function automatically estimates initial values for a Logistic equation): Github–Local

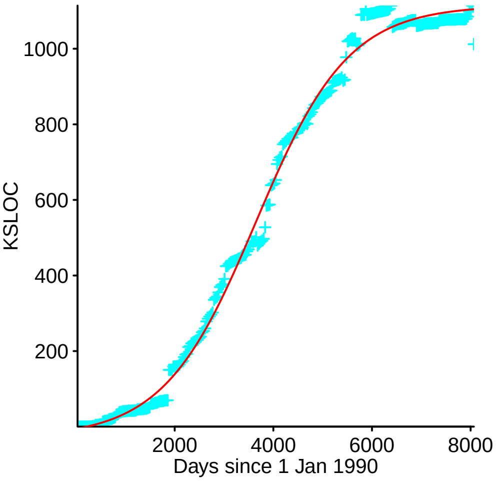


Figure 11.52: Lines of code in the GNU C library against days since 1 January 1990. Data from González-Barahona.[696] Github–Local

Plugging the fitted model coefficients into the Logistic equation give:

$$KLOC = -28 + \frac{1115-(-28)}{1+e^{(3652-Days)/935}}$$

Since these measurements were made, the C Standard's committee, JTC1 SC22/WG14, have started work on revising the existing specification; the model's prediction that glibc will max out at around 1,115,000 lines is unlikely to remain true for many more years.

A study by Chen, Groce, Fern, Zhang, Wong, Eide and Regehr[337] investigated fault experiences in a C compiler and JavaScript engine, by having them process randomly generated programs. Some programs failed to be correctly processed (1,298 in gcc and 2,603 in Mozilla's SpiderMonkey), and many of these failures could be traced back to the same few underlying mistakes in the code, i.e., some fault experiences were encountered more often than others. Figure 11.53 shows the number of failing programs that could be traced back to the same mistake, the curved green line is a regression fit (a biexponential, or double exponential); the two straight lines are the exponentials that are added to form the bi-exponential.

The `nls` has a `SSbiexp` starter function, which performs poorly for this data (or, at least, your author could not make it do well).

The sample contains count data, with many very small values, implying a Poisson error distribution. The `gnm` function, in the `gnm` package, has an option to select an error distribution.

The formula notation used by `gnm` is based on function calls,[1835] rather than the binary operators used by `glm` and `nls`. The formula argument in the following call (used to fit the model plotted in figure 11.53), contains two exponentials (specified using the `instances` function), the literal `1` is a placeholder for an unknown constant multiplied (the `Mult` function) by an exponential (the `Exp` function); as with calls to `nls`, starting values are required:

```
library("gnm")

fail_mod=gnm(count ~ instances(Mult(1, Exp(ind)), 2)-1,
                data=wrong_cnt, verbose=FALSE,
                start=c(2000.0, -0.6, 30.0, -0.1),
                family=poisson(link="identity"))
```

See fig 6.25 for a discussion of one possible reason the biexponential is such a good fit.

Various natural processes can be modeled using a sum of (possibly) many exponentials, and specific techniques have been created to fit data to this specific non-linear case; some of these technique have the advantage of being able to operate with a very approximate starting estimates of the exponent.

The mexpfit function in the pracma package implements one such technique. Support is rudimentary, at the time of writing, but mexpfit can save a lot of time by providing a workable estimate for a call to gnm.

```
library("pracma")

me_mod=mexpfit(wrong_cnt$ind, wrong_cnt$count, p0=c(-0.9, -0.1))
print(me_mod)  # no summary support, at the time of writing
```

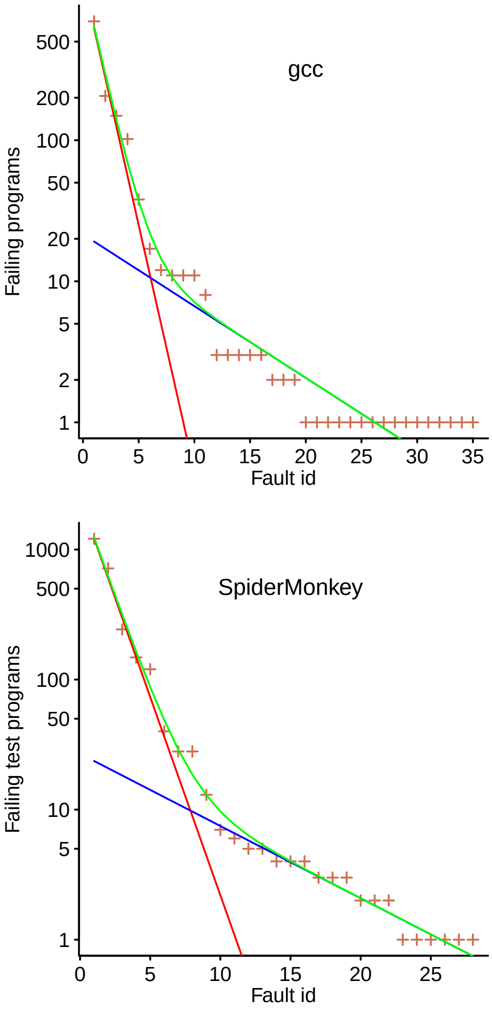


Figure 11.53: Number of failing programs caused by unique fault experiences in gcc (upper) and SpiderMonkey (lower). Fitted model in green, with two exponential components in red and blue. Data kindly provided by Chen.[337] Github–Local

11.5.1 Power laws

Plotting values drawn from a power law distribution using a log scale for both axis, produces a straight line. This straight line characteristic is not unique to power laws, it can also appear to occur with samples drawn from other distributions, e.g., an exponential distribution (see section).[xxiv]

The poweRlaw package includes functions for fitting and checking whether a power law is likely to be a good fit for a sample.[363]

When the model being fitted contains one explanatory variable, thought to have the form of a power law, functions from the poweRlaw package can be used. However, this package does not support more complicated models, and so other regression modeling functions have to be used when a power law is one of multiple components in a model, e.g., nls.

A study by Queiroz, Passos, Valente, Hunsen, Apel and Czarnecki[1525] analysed the conditional compilation directives (e.g., #ifdef) used to control the optional features in 20 systems written in C. Researchers in this area use the term *feature constant* to denote macro names used to control the selection of optional features and *scattering degree* to describe the number of ifdefs that refer to a given feature constant, e.g., if the macro SUPPORT_X appears in two ifdefs, it has a scattering degree of two.

Figure 11.54 shows the total number of feature constants (y-axis) having a given scattering degree (x-axis) in these 20 systems, lines are a power law (red) and exponential (blue) of fitted models; the numbers are the p-values for the fit (higher is better, i.e., fail to reject the hypothesis). This analysis is a fishing expedition involving 20 systems, and a power law is suggested by the visual form of the plotted data; with multiple tests it is necessary to take into account the increased likelihood of a chance match.

If 0.05 is taken as the p-value cutoff, for one test, below which the distribution hypothesis is rejected, then $(1 - 0.95^{20}) \rightarrow 0.64$ is the cutoff when 20 tests are involved. Some systems have p-values above the cutoff for one of the power law or exponential fitted models, and so the given distribution is not rejected for these systems.

The poweRlaw package supports discrete and continuous forms of heavy tailed distributions. The scattering degree is an integer value, and the following code fits both a discrete power law and exponential to the data (the continuous forms are conpl and conexp respectively):

[xxiv] Papers[1151] claiming to have found a power law, purely on the basis of a plot showing points scattered roughly along a straight line, are a common occurrence.

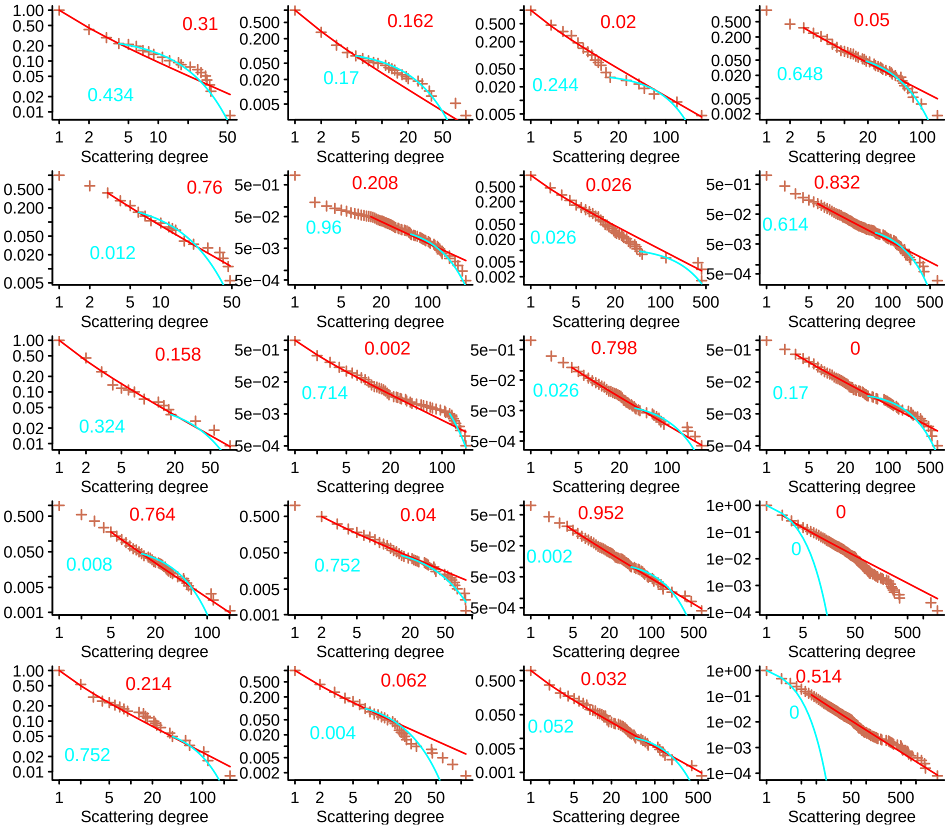


Figure 11.54: Power law (red) and exponential (blue) fits to feature macro usage in 20 systems written in C; fail to reject p-value for 20 systems is 0.64. Data from Queiroz et al.[1525] Github–Local

```
library("poweRlaw")

# Fit scattering degree
# displ is the constructor for the discrete power law distribution
pow_mod=displ$new(FS$sd)
exp_mod=disexp$new(FS$sd)  # discrete power exponential

# Estimate the lower threshold of the fit
pow_mod$setXmin(estimate_xmin(pow_mod))
exp_mod$setXmin(estimate_xmin(exp_mod))

# Plot sample values
plot(pow_mod, col=point_col, xlab="Scattering degree", ylab="")
lines(pow_mod, col=pal_col[1]) # Plot fitted line
lines(exp_mod, col=pal_col[2])

# Bootstrap to test hypothesis that sample drawn from a power law
bs_p=bootstrap_p(pow_mod, threads=4, no_of_sims=500)
text(40, 0.5, bs_p$p, pos=2, col=pal_col[1]) # Display value
```

The power law equation includes a minimum value of x, scattering degree in this case, below which it does not hold. The `estimate_xmin` function estimates the value, x_{min}, that minimises the error between the fitted model and the data. The `new` function, called by the constructor, sets x_{min} to the minimum value present in the data. It is common for power laws to fit a subset of the data.

11.6 Mixed-effects models

Mixed-effects models are used to model measurements of multiple correlated measurements of the same subjects (e.g., before/after measurements of the same subject), and clusters of related subjects. The regression techniques discussed so far assume that measurements are not correlated with each other.

In a mixed-model the explanatory variables are classified as either a *fixed-effect*, or a *random-effect* (sometimes called a *covariate*). Technically the effects are not fixed and are not random[xxv]. One way to think about classifying the two kinds of explanatory variables, is to look at the impact they have on the response variable:

- fixed effects influence the mean value of the response variable, and are associated with the entire population,
- random effects influence the variance of the response variable, and are associated with individual subjects.

A study by Balaji, McCullough, Gupta and Agarwal[120] measured the power consumption of six different Intel Core i5-540M processors executing the SPEC2000 benchmark at various clock frequencies; the six processors are a sample of the entire population of Intel Core i5-540M processors. The power consumption characteristics might be modeled by combining the data from all six processors; the following is the `summary` output for this model: Github–Local

```
Call:
glm(formula = meanpower ~ frequency, data = power_bench)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.5746  -0.1882   0.0413   0.1902   2.2965

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  2.12594    0.01506   141.2   <2e-16 ***
frequency    1.95248    0.00767   254.6   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 0.1429928)

    Null deviance: 10692.7  on 9980  degrees of freedom
Residual deviance:  1426.9  on 9979  degrees of freedom
AIC: 8916.2

Number of Fisher Scoring iterations: 2
```

This model does not provide any information about how performance varies between processors. The identity of the processor measured could be included in the model (see Github–regression/hotpower-proc.R), but this is a model of the sample, and it cannot be used to deduce anything about the population from which it was drawn.

How might the sample of processors be modeled in a way that provides an estimate of population variability? Possible techniques include:

- building a regression model for each processor, and average these six models in some way, e.g., use the coefficients from each of the six models to build a regression model that is a model of models,

  Electronic circuit theory tells us that processor power consumption is proportional to clock frequency, and figure 11.55 shows the results of fitting a separate straight line to the data for each processor.
- building a *mixed-effects model*. A mixed-effects model (also known as a *hierarchical model*) might be viewed as a model of models; mathematically it uses a more direct approach, making more effective use of the available data than the method described above.

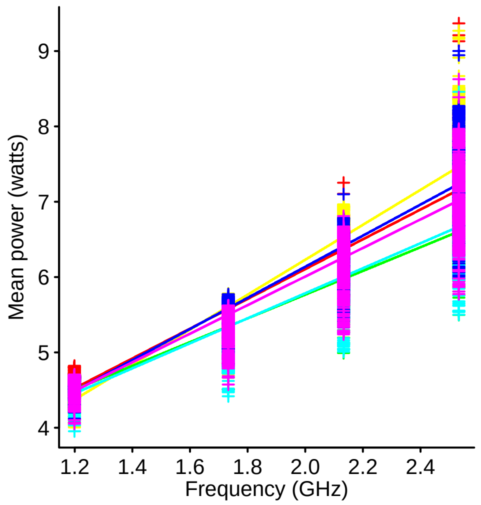


Figure 11.55: Power consumption of six different Intel Core i5-540M processors running at various frequencies; colored lines denote fitted regression models for each processor. Data from Balaji et al.[120] Github–Local

A number of different packages are available for fitting mixed-effects models, this book uses `lme4`, whose workhorse functions are the `glmer` and `lmer` functions.[xxvi]

The `lme4` package extends the formula notation to support the specification of random effects. In the following code:

```
library("lme4")
```

[xxv] Some authors point this out, and then proceed to use what they consider to be more technically correct terms, this book follows common usage because it is common; these terms crop up as named parameters in functions, and appear in output information.

[xxvi] A call to the `glmer` function that uses the default family distribution, i.e., gaussian, generates a warning that this usage is deprecated and `lmer` should be used.

```
# Express in Gigahertz (otherwise lmer does not converge)
power_bench$frequency=power_bench$frequency/1000000

p_mod=lmer(meanpower ~ frequency + (1 | processor), data=power_bench)
p_mod=lmer(meanpower ~ frequency + (frequency-1 | processor), data=power_bench)
p_mod=lmer(meanpower ~ frequency + (frequency   | processor), data=power_bench)
p_mod=lmer(meanpower ~ frequency + (1 | processor) + (frequency-1 | processor),
                                                              data=power_bench)
```

- first call to lmer: frequency is the fixed-effect and (1 | processor) is the random-effect; the 1 specifies variation in the intercept value, and the source of this variation is the processor variable (i.e., the column having this name in the data frame). When plotted the model might look something like the upper plot of figure 11.56, with six lines intersecting the y-axis at different points, but all having the same slope,
- second call to lmer: (frequency-1 | processor) specifies there is variation in the slope, and the source of this variation is the processor variable (this can also be written as: (frequency+0 | processor)). When plotted the model might look like the lines in the middle of figure 11.56, where all lines intersect the y-axis at the same point but have different slopes,
- third call to lmer: (frequency | processor) specifies that variation in the processor variable may cause both the intercept and the slope to vary, and the intercept and slope are correlated (can also be written as: (1+frequency | processor)). When plotted, the models might look like the lines in the lower plot of figure 11.56, where the lines have different intersections and slopes,
- fourth call to lmer: the operands (1 | processor)+(frequency-1 | processor) differs from the third call in that the intercept and slope are not correlated.

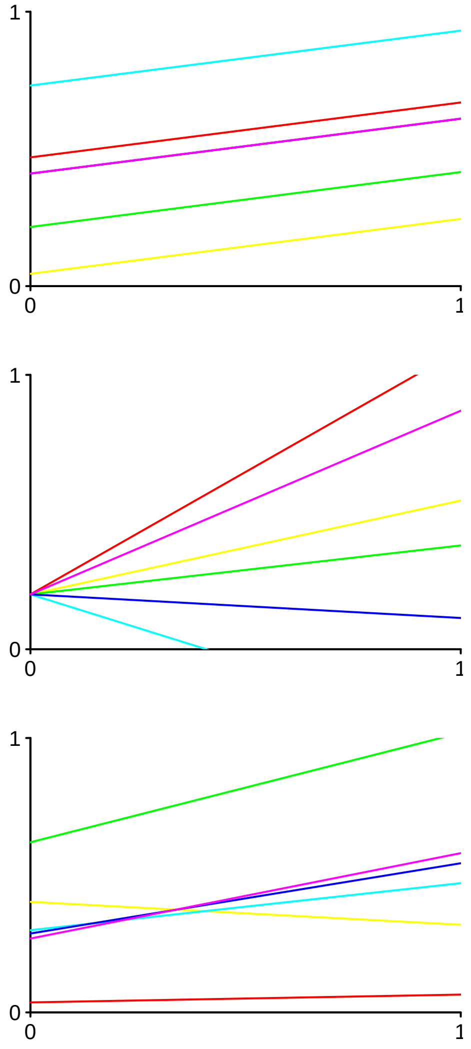


Figure 11.56: Example showing the three ways of structuring a mixed-effects model, i.e., different intersections/same slope (upper), same intersection/different slopes (middle) and different intersections/slopes (lower). Github–Local

The following is the summary output from a mixed-effects model, where the processor is a random-effect on both the intercept and slope: Github–Local

```
Linear mixed model fit by REML ['lmerMod']
Formula: meanpower ~ frequency + (frequency | processor)
   Data: power_bench

REML criterion at convergence: 6300.3

Scaled residuals:
    Min      1Q  Median      3Q     Max
-4.0533 -0.4866  0.1453  0.4994  6.6744

Random effects:
 Groups    Name        Variance Std.Dev. Corr
 processor (Intercept) 0.13202  0.3634
           frequency   0.07552  0.2748   -0.99
 Residual              0.10941  0.3308
Number of obs: 9981, groups:  processor, 6

Fixed effects:
            Estimate Std. Error t value
(Intercept)   2.1740     0.1490   14.59
frequency     1.9156     0.1124   17.04

Correlation of Fixed Effects:
          (Intr)
frequency -0.993
convergence code: 0
Model failed to converge with max|grad| = 0.0314161 (tol = 0.002, component 1)
```

The values for (Intercept) and frequency, listed under Fixed effects:, are very similar to the combined data model fitted earlier. Annoyingly, the summary output does not include p-values. These can be obtained using the Anova function from the car package.

The Random-effects: table lists the variation introduced by processor (listed in the Groups column, on the variables listed in the Name column); the Std.Dev. column lists the estimated standard deviation in the corresponding coefficient listed in the Fixed eff ects: table. Residual lists the residual random effects left after taking into account all the specified random-effects.

As an example, taking frequency, there are two sources of uncertainty in its contribution to the response variable (as expressed in its model coefficient), one from fixed-effects, and a random-effect caused by the variation between processors.

Plotting the 95% confidence intervals, for the intercept and slope of a mixed-effects model, provides a visualization of the relative contribution of the sources of variation. Figure 11.57 was generated using the following code, with data from the six processors:

```
library("lattice")
library("lme4")
library("gridExtra")

proc_mod=lmer(meanpower ~ frequency +(frequency | processor),
                                                data=power_bench)
dp_orig=dotplot(ranef(proc_mod, condVar=TRUE), main=FALSE)

power_bench$shift_freq=power_bench$frequency-min(power_bench$frequency)
proc_mod=lmer(meanpower ~ shift_freq +(shift_freq | processor),
                                                data=power_bench)

dp_shift=dotplot(ranef(proc_mod, condVar=TRUE), main=FALSE)

# dotplot comes from the lattice package, which uses grid layout
grid.arrange(dp_orig$processor, dp_shift$processor, nrow=2)
```

Figure 11.57, upper plot, is the model fitted using the original data; the intercept (upper left) and slope (upper right) appear to be correlated. Looking at the straight line fits for each processor in figure 11.55, they appear to share an origin starting at the lowest frequency measured; an intercept included as a random effect has a common origin assumed to start at zero (see figure 11.56). Shifting frequency values down, by the minimum measured value, and refitting a model produces the confidence intervals in the lower plot. The correlation has disappeared; perhaps including the intercept as a random effect is not worthwhile.

Refitting a model without the intercept as a random effect, produces a model that differs from previous models by a small amount (see Github–regression/hotpower-mix-plot).

There is an upper limit on the number of random effects (i.e., number of unknowns) that can occur in a model. The total number of unknown random effects must be less than the number of observations, otherwise the equations do not have a unique solution. A continuous explanatory variable counts as a single unknown, while a variable holding nominal or ordinal values contributes one unknown for each of the possible discrete values (there is no slope associated with fitting a variable that is not treated as being continuous).

The bootstrap can be used to calculate confidence intervals for a mixed-effects model.

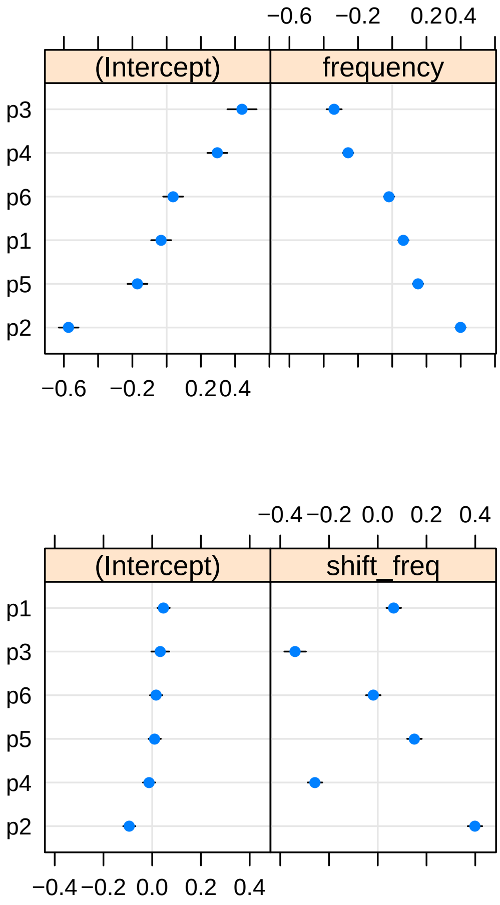


Figure 11.57: Confidence intervals, 95%, for first (upper) and second (lower) call to lmer; within-subject intercepts (left column) and slopes (right column) for the mixed-effects models in the adjacent code. Github–Local

11.7 Generalised Additive Models

The regression modeling techniques discussed so far have required the analyst to specify an equation expressing the detailed relationship between explanatory variables and the response variable (these are said to be *parametric models*). If no equation provides a reasonable fit, or accuracy of prediction is important (rather than understanding), then a *Generalised additive model* (GAM) is an alternative approach. A GAM only requires a list of explanatory variables and a response variable to be specified (these are said to be *nonparametric models*).

A GAM is built by finding the best fit for a sequence of polynomial equations (e.g., some form of spline), that smoothly captures the shape of the data. These smooth equations might be used to make predictions, or when the fitted model is plotted may suggest possible parametric equations. The details of the fitted equations are not a source of understanding, but they may make good predictions.

The gam function, in the mgcv package, can be viewed as extending the functionality of glm to support a variety of nonparametric smoothing functions (the gam package is simpler, but does not offer such a wide range of functionality). The following code shows formulas using a potentially different smoothing polynomial for each explanatory variable (first line below), a different smoothing polynomial for some combinations of explanatory variables (second and third line), a combination of a smoothing polynomial and parameterised form (fourth line), or an interaction between a smoothed and non-smoothed variable (fifth line; the by parameter, rather than the : operator is used):

```
mod=gam(y ~ s(x_1) + s(x_2) + s(x_3), data=foo_bar)
mod=gam(y ~ s(x_1) + s(x_2, x_3), data=foo_bar)
mod=gam(y ~ s(x_1) + s(x_2, x_3) + s(x_3, x_4) + s(x_4), data=foo_bar)
mod=gam(y ~ x_1 + s(x_2) + x_3, data=foo_bar, family="poisson")
mod=gam(y ~ x_1 + s(x_2, by=x_1) + x_3, data=foo_bar, family="poisson")
```

The smoothing function, s, supports a variety of options for controlling the fitting process; two that are likely to be encountered are k, which specifies an upper limit on the degrees of freedom that can be used in the fitted polynomial, and bs, a string identifying the kind of smoother (e.g., "tp", the default, for a thin plate regression spline and "cr" for a cubic regression spline).

The value of k needs to be large enough to support the degrees of freedom needed by a polynomial capable of representing the underlying pattern in the data; the gam.check function provides information about fitted models that can be used to help select a value for k.

The fitting procedure used, by the mgcv version of gam, tries to avoid overfitting by making every degree of freedom pay its way (using, for instance, *penalized regression splines*). Criteria used for measuring the *cost-effectiveness* of more complicated models include generalised cross-validation (GCV; the default) and AIC. The select argument provides support for *null space penalization*, see package documentation for details.

A study by Lee and Brooks[1091] built a model to predict the performance and power consumed by applications running on processors having various hardware configurations, e.g., number of registers, size of cache and instruction latency.

The following additive model is based on the one proposed by Lee et al, and explains over 95% of the variance in the data (see Github–regression/lee2006.R). While this model is likely to be useful for prediction, it provides virtually no insight into the impact of various hardware attributes on performance characteristics.

```
l_mod=gam(sqrt(bips) ~ benchmark + fix_lat
                           +s(depth, k=4) + s(gpr_phys, k=10)
                           +s(br_resv, k=6) + s(dmem_lat, k=10) +
                                                   s(fpu_lat, k=6)
                           +s(l2cache_size, k=5) + s(icache_size, k=3) +
                                                 s(dcache_size, k=3)
                           +s(depth, gpr_phys, k=10)+s(depth, by=width, k=6)
                           +s(gpr_phys, by=width, k=10)
                   , data=lee)
```

The analysis associated with figure 8.33 used two approaches to modeling the number of accesses to a function's local variables. Without knowing anything about what relationships might exist between explanatory and response variables, and being willing to use very high degree polynomials, it is possible to build and use gam to build a prediction model.

In the calls to gam below, the first assumes there is an interaction between the two explanatory variables (allowing up to 75 degrees of freedom), and the second assumes the variables are independent (allowing up to 50 degrees of freedom for each of them). While the fitted model might make usable predictions (see Github–sourcecode/local-use/obs-fit.R), the use of such high degree polynomials suggests that the underlying processes have a non-polynomial form.

```
locg_mod=gam(norm_occur ~ s(object.access, total.access, k=75),
                             data=common_loc, family=Gamma)

locp_mod=gam(norm_occur ~ s(object.access, k=50)+s(total.access, k=50),
                             data=common_loc, family=Gamma)
```

11.8 Miscellaneous

Topics that your author has had to deal with, from time to time.

11.8.1 Advantages of using lm

This book promotes glm as a one-stop solution, however, the lm function has some advantages over glm, including:

- requiring less cpu time to fit a model. If many models need to be fitted on a regular basis, the performance difference may be worth considering,
- requiring less memory to fit a model. For extremely large datasets, memory requirements may be excessive for glm; possible solutions that continue to use glm are discussed below,
- the algorithm used by lm is always guaranteed to converge to a solution, singularities generated by correlation between explanatory variables excluded. There are edge cases where glm does not find a solution without being given some reasonable starting values.

The implementation of lm is based on the mathematics of *Ordinary Least Squares* (OLS), and the data has to satisfy additional conditions for OLS to be applicable. Perhaps the most important new condition is that the error variance in the measurements be constant (in practice close to constant is usually good enough). The ncvTest function, in the car package, checks that a fitted model meets this requirement; the spreadLevelPlot function provides some visualization; also, see the lmtest function.

A user interface issue with models fitted using glm is that they do not come with a scale-invariant goodness of fit number, i.e., the R-squared value.

11.8.2 Very large datasets

The biglm package supports fitting regression models using data that is too large to fit in memory all at once; the models are built using an incremental algorithm, which only requires a subset of the data to be held in memory at any time. A variety of options are available for creating chunks of data to feed into the model building process, including incremental reading from files and databases.

The biganalytics package extends the bigmemory package by providing interfaces to various analytic packages, such as biglm (see Github–benchmark/bounds_chk.R).

11.8.3 Alternative residual metrics

The error metric used by many regression techniques is based around squaring the difference between the actual and predicted value. This choice has been driven by the theoretical usefulness of the mathematical properties of sum-of-squares. Other error metrics are available to fit models, e.g., the absolute difference between actual and predicted values.

The rlm function, in the MASS package, supports analyst specified functions for calculating the residual to be minimised when fitting a model. The robustbase and robust packages support a wide variety of functionality.

11.8.4 Quantile regression

The techniques discussed up to this point are based around predicting the expected value of the mean. Quantile regression is based on the proportion of data points above/below the fitted equation; it is robust to the presence of outliers, and is not influenced by the form of the error distribution.

The rq function in the quantreg package fits quantile regression models. Figure 11.58 was generated using the following code (also see fig 8.13):

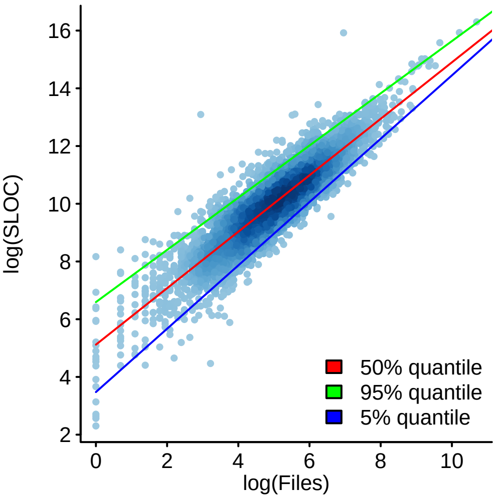


Figure 11.58: Number of files and lines of code in 3,782 projects hosted on Sourceforge; lines are 95%, 50% and 5% quantile regression fits. Data from Herraiz.[1785] Github–Local

```
library("quantreg")

quant_fit=function(tau_val, col_str)
{
rq_mod=rq(log(SLOC) ~ log(Files), data=proj_inf, tau=tau_val) # tau is the quantile
pred=predict(rq_mod, newdata=data.frame(Files=x_bounds))

lines(log(x_bounds), pred, col=col_str)

return(rq_mod)
}

plot(log(proj_inf$Files), log(proj_inf$SLOC),
        col=densCols(log(proj_inf$Files), log(proj_inf$SLOC)), pch=20,
        xlab="log(Files)", ylab="log(SLOC)\n")

x_bounds=exp(seq(0, log(1e5), by=0.1))

rq05_mod=quant_fit(0.05, pal_col[3]) # specify quantile and color
rq50_mod=quant_fit(0.5, pal_col[1])
rq95_mod=quant_fit(0.95, pal_col[2])
```

A line fitted to the 50% quartile has half the measurement points below/above it, while the 95% quartile line divides the measurements such that 95%/5% are below/above (the division of measurements for the 5% quartile is reversed).

11.9 Extreme value statistics

Extreme value statistics deals with the probability of occurrence of extreme values, e.g., use of maximum memory available memory, or minimum response time. The two main techniques are Generalized Extreme Value (GEV) and Generalized Pareto (GP); the `extRemes` package supports both techniques.

The GEV approach analyses each maximum value that occurs in a specified interval (e.g., maximum daily fault reports within each month), while the GP approach analyses all values above a specified threshold value (e.g., all program runs taking longer than x seconds). The equation fitted by each approach both contain three parameters: μ (the mean value for GEV, and the threshold for GP), σ a multiplier that scales the function, and ξ (greek lower-case xi) a shape parameter (depending on whether ξ is equal/greater/less than zero, the equations simplify to more well-known distributions; Gumbel, Fréchet and Weibull respectively for GEV, and Exponential, Pareto and Beta for GP).

Some of the WG21 (the ISO C++ Standard working group) email reflectors receive a lot of traffic, particularly the Core and Lib reflectors. What is the maximum number of messages on one day that is likely to occur within a 10-year period?

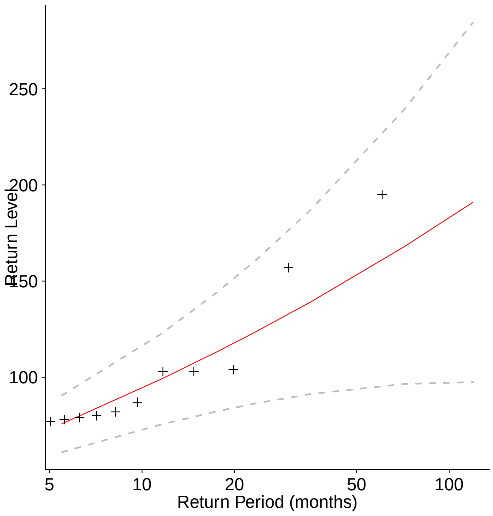


Figure 11.59: Expected maximum number of daily emails to the C++ lib email list expected to occur within a given period of months, with 95% confidence intervals; a GEP fitted model. Data kindly extracted from the WG21 mailing list archive by Roger Orr.

Roger Orr[xxvii] kindly extracted the date of every message posted since February 2016 (configuration changes over the years make it non-trivial to obtain data before this date) to the Core and Lib mailing list.

The `fevd` function, in the `extRemes` package, calculates the parameters for the extreme value distribution that best fits the data. When using GP a threshold has to be chosen, and the `threshrange.plot` can be used to help select a value.

The default value of options assume stationary data (i.e., the mean does not change over time); an equation can be given for each model parameter specifying how it changes with time.

```
library("extRemes")

max_mod=fevd(month_max$V1, type="GEV", period.basis="month")
plot(max_mod, rperiods=c(6, 12, 18, 36, 72, 120), type="rl", col="red", main="")

summary(max_mod)
```

[xxvii] Roger is the convenor of the UK's BSI C++ panel.

Figure 11.59 shows a GEP fitted model for the maximum number of daily emails expected to occur (y-axis) within a given number of months (x-axis), for WG21's the Lib email list; the pluses are actual occurrences, and dashed lines 95% confidence intervals.

The model used is very simplistic, and does not take into account the growth in members joining these lists and traffic lost when a new mailing list is created for a new committee subgroup.

11.10 Time series

Time series analysis deals with measurements that are sequentially correlated. An example of correlated measurements is current room temperature, which is likely to be similar to the temperature 10 minutes ago, and the temperature 10 minutes from now. Techniques developed to analyse time-series can be used to analyse measurements of any quantity, where a correlation exists between successive measurements.

The base system provides basic functions for analyzing time series of continuous values.

A time series contains one or more of the following three components:

- underlying trend: which changes slowly,
- regular recurring pattern of changes (known as *seasonality*): for instance, expected daytime temperature throughout the year,
- random, irregular or fluctuating component.

The `stl` function (Seasonal Trend using Lowess) provides a way of splitting a time series into these three components (the argument must be an object of type `ts`, with a user specified frequency; the `stl` function does not automatically detect the recurrence period), and there is a corresponding `plot` function.[xxviii]

Figure 11.60, from a study by Eyolfson, Tan and Lam,[558] shows the three time-series components of the hourly rate of commits to the Linux kernel source tree, over the days of a week (the commits during the same hour of the same day were summed). The `stl` function assumes a fixed, recurring, pattern of seasonal behavior, a slowly changing trend, with everything else classified as random noise.

```
# A seasonal frequency has to be specified
hr_ts=ts(linux_hr, start=c(0, 0), frequency=24)
plot(stl(hr_ts, s.window="periodic"))
```

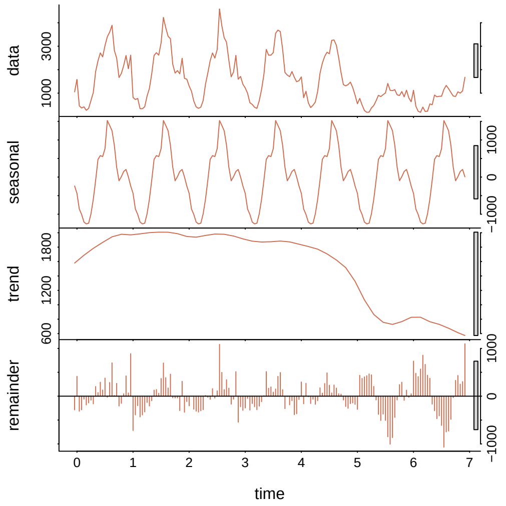


Figure 11.60: The three components of the hourly rate of commits, during a week, to the Linux kernel source tree; components extracted from the time series by `stl`. Data from Eyolfson et al.[558] Github–Local

Possible outputs from time-series analysis include:

- a fitted model specifying how the value of a quantity at time t depends on its values at earlier measurement times (often at $t-1$),
- a regression model, adjusted for the correlation between sequential measurements,
- a power spectrum showing the dominant frequencies present in the data,
- a hierarchical clustering of multiple time series,
- a list of patterns, motifs, that occur within a time series.

Structure is often added to the linear nature of time by imposing repeating fixed length intervals, such as hours of the day and days of the week. Many time series analysis techniques require measurements to be made at fixed length intervals; analysis of measurements at irregular intervals is not discussed here.

Some library functions use a time series datatype for representing time related measurements. The `ts` function, part of the base system, converts a vector to class `ts` (many time series functions will automatically convert vectors to this class).

The `xyplot` function, in the `lattice` package, can be used to create a time series strip chart, see fig 8.19.

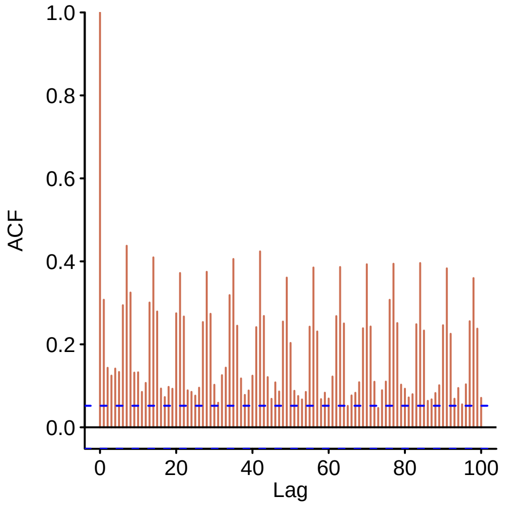


Figure 11.61: Autocorrelation of number of defects found on a given day, for development project C. Data kindly provided by Buettner.[269] Github–Local

[xxviii] The `decompose` function, part of the base system, implements the same functionality in a less sophisticated way.

11.10.1 Cleaning time series data

Many time series techniques implicitly assume that measurement data occurs at regular intervals. A measurement process may only record events when they occur and if no event occurred in within an interval there may be no data-point for that interval. The cleaning process includes ensuring that every interval contains a value (which may be zero or inferred from surrounding values).

A study by Buettner[269] gathered project staffing information for several commercial development software projects. On large commercial projects the amount of work done at weekends is likely to be zero (except for the weeks prior to major deliveries), and the autocorrelation of project activity is likely to show a recurring pattern involving two consecutive days separated by seven days, i.e., weekends and weekdays.

Figure 11.61 shows the autocorrelation of the number of defects found on a given day, for one development project. The seven-day recurring pattern contains a three consecutive day pattern, are the developers only working a four-day week? It turns out [xxix] that contractors on some projects work a two-week cycle, with extra hours worked one week and then not working the Friday of the following week. The extent to which regular staffing level differences, between Friday and other weekdays, has to be taken into account, will depend on the kind of analysis performed (weekends can be handled by excluding them from the analysis, focusing on where most effort occurs, i.e., week days).

Measurements made on public holidays, such as the New Year, are very likely to differ from normal work days. Removing public holidays from the data will scramble the association with day of the week. The extent to which day of the week is a more important factor in the analysis, than public holidays, has to be considered.

11.10.2 Modeling time series

The expected mean of a time series can be modeled using one or both of the following two approaches (series whose variance is serially correlated are discussed later):

- the *Autoregressive model* (AR), models the value at time t as a weighted combination of values from earlier time steps, plus some amount of added noise, w_t, for instance:

  $x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + w_t$

  is an autoregressive model of order 2, abbreviated AR(2); it is based on values going back two time steps (with weights ϕ_1 and ϕ_2).

  The `ar` function fits data to an autoregressive model.

- the *Moving Average model* (MA), models the value at time t as the sum of noise, w_t, and a weighted combination of the noise from earlier time steps, for instance:

  $x_t = w_t + \theta_1 w_{t-1}$

  is a moving average model of order 1, abbreviated MA(1); it uses a value from one time step back (with weight θ_1).

  The `arima` function, with the first two values of the `order` argument set to zero, fits data to a moving average model.

The autocorrelation function, `acf`, returns and plots the correlation of a time series with itself at successive lag intervals (i.e., the correlation of the measurement at time t with the measurement at time $t+n$; the default sequence of lags is `n=1:25`); see figure 11.62. For the AR(1) model, $x_t = \phi x_{t-1}$, the impact of serial correlation on values separated by k lags (time intervals) decreases by ϕ^k.

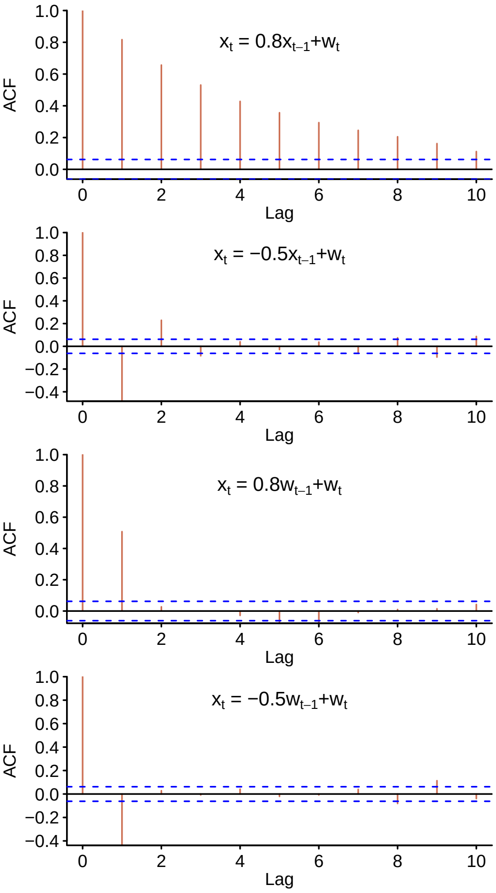


Figure 11.62: Autocorrelation of two AR models (upper plots) and two MA models (lower plots); the same models are used in figure 11.63. Github–Local

The lag 0 autocorrelation is always one, and the two dotted blue lines are 0.05 p-value bounds. Each lag is a hypothesis test, and with 25 hypothesis tests (the default) at least one calculated value is expected to exceed a 0.05 p-value with probability $1 - 0.95^{25} \rightarrow 0.72$; also, successive measurements are correlated, so neighbouring lag points are likely to show similar significance levels.

The partial autocorrelation function (the `pacf` function) calculates and plots the correlation at lag k, after removing the effect of any correlation generated by terms at shorter lags; see figure 11.63. The partial autocorrelation at lag k is the k^{th} coefficient of an AR(k) model.

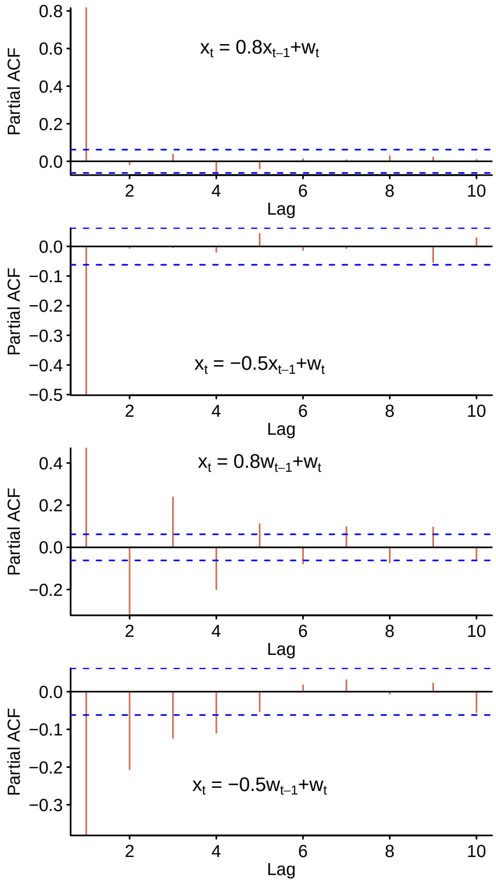


Figure 11.63: Partial autocorrelation of two AR models (upper plots) and two MA models (lower plots); the same models are used in figure 11.62. Github–Local

[xxix] Email discussion with Buettner.

The previous two plots illustrate how short range correlations in an AR model have a long range impact on the values returned by `acf`, but an MA model does not have a long range impact, while the opposite behavior is seen in the values returned by `pacf`. An ARMA model always behaves in the most unhelpful way.

An ARMA model (*Autoregressive Moving Average*) is a combination of an AR and MA model, e.g., ARMA(2, 1) is the sum of an AR(2) and MA(1) model, such as the following:

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + w_t + \theta_1 w_{t-1}$$

The `ARMAacf` function takes a specification of an ARMA model, and returns what `acf` would return when passed a time series following this model (the `pacf=TRUE` option switches the behavior to that of `pacf`).

A time series is said to be *stationary* if the expected mean value does not change over successive measurements, i.e., $E[t_i] = E[t_{i+k}]$. The mathematics behind both the basic AR and MA model fitting techniques assume a stationary time series (more sophisticated techniques are available; ARIMA (*Autoregressive Integrated Moving Average*) handles some non-stationary time series: supported by the `arima` function).

Many software engineering processes include non-stationary components, e.g., varying number of developers working on a project, increasing number of customers, system updates, etc.

Time series analysis techniques are not limited to measurements involving time, they can be applied to any data that has serial correlation between measurements.

A study by Hindle, Godfrey and Holt[823] investigated the indentation of the first non-whitespace character on a line, for code written in a variety of languages. Figure 11.64 shows the autocorrelation of a list, ordered by indentation, of the total number of lines having a given indentation.

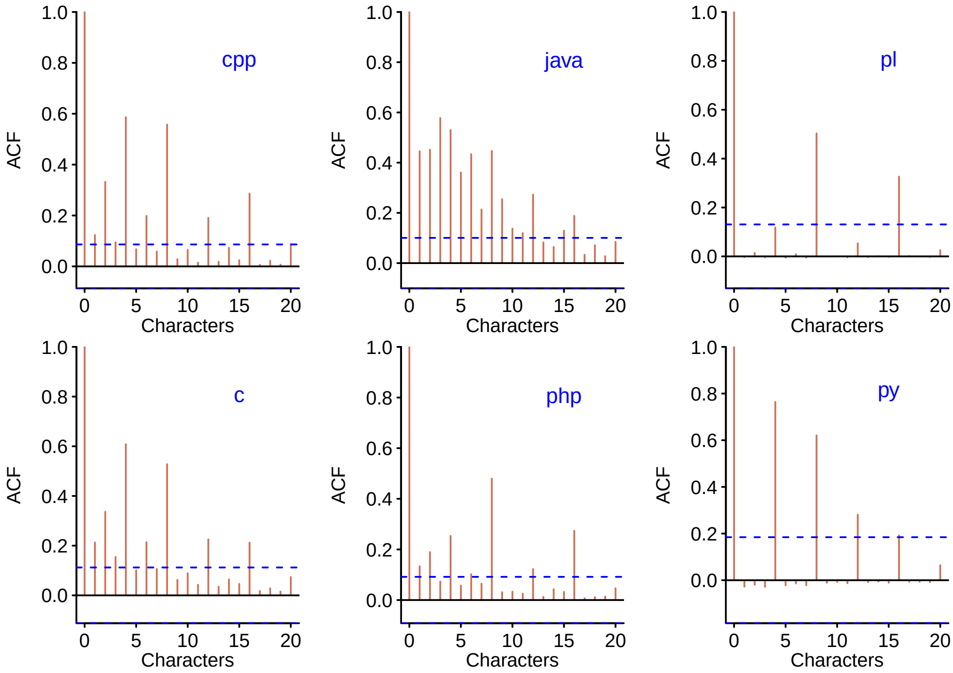


Figure 11.64: Autocorrelation of indentation of source code written in various languages. Data from Hindle et al.[823] Github–Local

11.10.2.1 Building an ARMA model

ARMA modeling takes as input a time series; if this time series is non-stationary, it has to be converted to a stationary form before model building can begin. Common reasons for a time series not being stationary and possible transforms to a stationary series include:

- a non-zero trend: for instance, the following equation contains an increasing time dependent trend:

  $$x_t = \alpha + \beta t + w_t$$

  Differencing can be used to remove trends, but care needs to be taken because this can introduce signals that are not in the original data. For instance, differencing the above equation gives:

  $$\Delta x_t = x_t - x_{t-1} = b + w_t - w_{t-1}$$

an MA(1) process, which the original series does not contain.

Subtracting the trend $\alpha + \beta t$ leaves just w_t; see the lower plot of figure 11.65,

- non-constant variance (known as *volatility* in the analysis of financial time series).

  If the growth in variance, over time, approximately follows the growth of the mean (i.e., a relatively consistent percentage change at each time step, e.g., $y_t = (1 + x_t)y_{t-1}$), then a log transform produces a time series with approximately constant variance (i.e., $\Delta(\log y_t) \approx x_t$, assuming $\log(1 + x_t) \approx x_t$).

+ A log transform requires special processing of any zero values; possible solution include adding a small amount to every value [xxx] and setting non-finite log-transformed values to zero (both have some impact on a fitted regression model). The 7digital data has increasing variance (more developers are employed and time to implement features decreases), and many zeroes; see Github–time-series/agile-day-starts.R.

- seasonality: this is a cyclic trend, e.g., changes recurring every year. Implementations of ARMA often include support for including a seasonal component in the model, e.g., the `seasonal` to the `arima` function,

The Augmented Dickey-Fuller test is a well known technique for checking whether a time series is stationary, others include the Phillips-Person test and the KPSS-test (supported by the `adf.test`, `pp.test` and `kpss.test` functions in the `tseries` package). These tests all have low power (i.e., fail to detect that a time series is non-stationary; they all fail to detect that the untransformed 7digital data is not stationary, see Github–time-series/agile-day-starts.R), and sometimes give contradictory results.

The plots produced by `acf` and `pacf` provide useful information about the likely structure and order of an ARMA model.

- if the plot produced by `acf` shows a decreasing trend, while the `pacf` shows a sharp cut-off (see figure 11.62), an AR model is a good place to start,
- if the plot produced by `acf` shows a sharp cut-off, while the `pacf` shows a decreasing trend (see figure 11.63), an MA model is a good place to start,
- if both plots show a decreasing trend, then some combination of AR and MA model is likely to be needed.

```
lwd=log(weekdays+1e-1)  # handle days with zero values

acf(lwd, xlab="Lag (working days)")
pacf(lwd, xlab="Lag (working days)")
```

Models fitted, by calling `arima` with various values of its `order` argument, can be compared using AIC (`arima` only returns the series mean, when a difference value of zero is passed to `order`; the `Arima` function, in the `forecast` package, is not limited in this way). The `auto.arima` function, in the `forecast` package, can be used to automatically find ARIMA model values that minimise AIC.

```
library("forecast")

arima(lwd, order=c(5, 0, 1))
auto.arima(lwd, max.order=7)
arima(lwd, order=c(1, 0, 1))
```

The two best fitting models, for the feature start data, are ARMA(5, 1) and ARMA(1, 1). The output from the last call to `arima` above is: Github–Local

```
Call:
arima(x = lwd, order = c(1, 0, 1))

Coefficients:
         ar1      ma1  intercept
      0.9627  -0.7993      0.561
s.e.  0.0158   0.0380      0.241

sigma^2 estimated as 1.789:  log likelihood = -1465.67,  aic = 2939.35
```

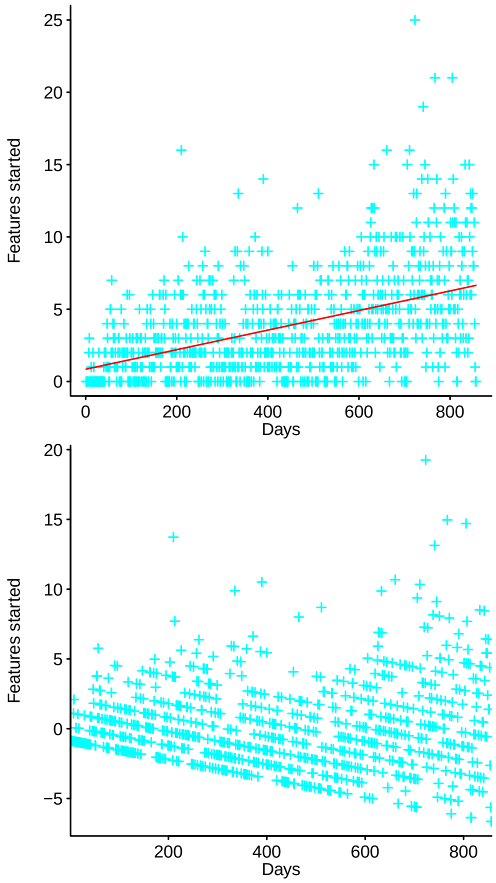


Figure 11.65: Number of features started for each day and fitted regression trend line (upper) and number of features after subtracting the trend (lower). Data kindly supplied by 7Digital.[1] Github–Local

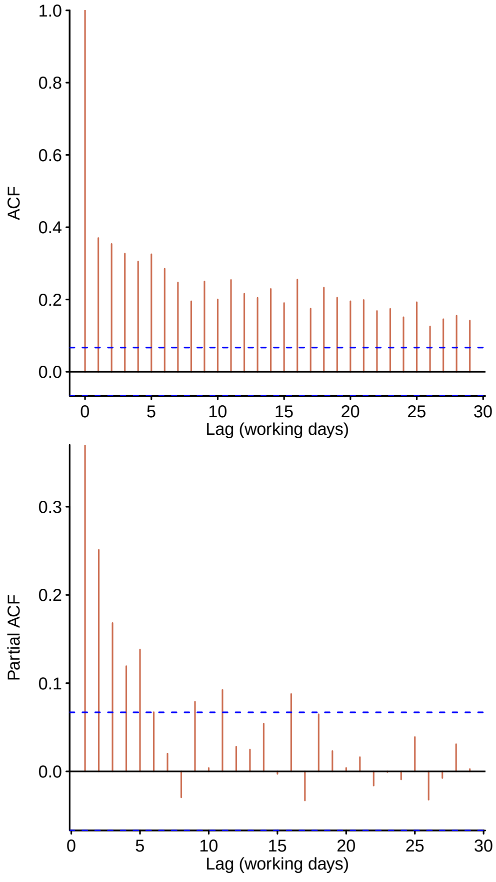


Figure 11.66: Autocorrelation (upper) and partial autocorrelation (lower) of the number of features started on a given day (after differencing the log transformed data), over the entire period of the 7digital data. Data kindly supplied by 7Digital.[1] Github–Local

[xxx] The value should not be so small that its log is a large negative value.

The `Coefficients:` table lists the model coefficients and their standard error. The `intercept` column is the mean value of the time series. The equation for one of the models is:

$x_t - 0.5610 = 0.9627(x_{t-1} - 0.5610) + w_t - 0.7344w_{t-1}$, which simplifies to:

$x_t = 0.5610(1 - 0.9627) + 0.9627x_{t-1} + w_t - 0.7993w_{t-1}$

the constant (log transformed) increment per time step is: $0.5610(1-0.9627) \rightarrow 0.0209253$.

Which of these two models provides the better explanation of the data? Features take different amounts of time to implement, and work can only start on a new feature when enough people have been freed up, through completion of work on other features. The coefficients of the AR component, of the ARMA(5, 1) model, can be interpreted as a probability that people working on a feature started a given number of days earlier will become available to start work on a new feature (see table 11.4).

	AR	Duration
ar1	0.19	0.32
ar2	0.11	0.16
ar3	0.09	0.11
ar4	0.07	0.07
ar5	0.10	0.05

Table 11.4: AR coefficients of ARMA(5, 1) model and percentage of features taking a given number of days to implement. Data kindly supplied by 7Digital.[1] Github–Local

This may be a just-so story, but stories are useful tools, but your author cannot think of one for the ARMA(1, 1) model.

Handling seasonal trends A seasonal ARIMA model can include AR, difference and MA components at an offset equal to the number of measurement intervals in the season. By default, the `auto.arima` function, in the `forecast` package, will return seasonal components (if any are found). The `seasonal` option can be used to specify seasonal components to the `arima` function.

The following code estimates a seasonal ARIMA model, for hourly commits to the Linux kernel source tree (see fig 11.60):

```
library("forecast")

hr_ts=ts(linux_hr, start=c(0, 0), frequency=24)

auto.arima(hr_ts)
arima(linux_hr, order = c(2,1,1), seas = list(order = c(1,0,1), period=24))
```

The coefficients of the first and second fitted models, below, differ because of differences in the algorithms used by the functions that fitted them, but are within each other's standard error (the third set of coefficients is for a slightly simpler model): Github–Local

```
Series: hr_ts
ARIMA(2,1,2)(1,0,0)[24] with drift

Coefficients:
         ar1      ar2     ma1    ma2    sar1    drift
      -0.892  -0.5348  0.5087  0.221  0.6751  13.0240
s.e.   0.244   0.1621  0.2694  0.223  0.0708  50.1094

sigma^2 estimated as 143870:  log likelihood=-1233.06
AIC=2480.12   AICc=2480.83   BIC=2501.95

Call:
arima(x = linux_hr, order = c(2, 1, 1), seasonal = list(order = c(1, 0, 1),
    period = 24))

Coefficients:
          ar1      ar2     ma1    sar1     sma1
      -0.8124  -0.2862  0.4483  0.8909  -0.4516
s.e.   0.2995   0.1177  0.3058  0.0546   0.1404

sigma^2 estimated as 129070:  log likelihood = -1229.02,  aic = 2470.03
```

```
Call:
arima(x = linux_hr, order = c(2, 1, 0), seasonal = list(order = c(1, 0, 1),
    period = 24))

Coefficients:
          ar1      ar2    sar1     sma1
      -0.3632  -0.1174  0.8980  -0.4727
s.e.   0.1007   0.0964  0.0519   0.1391

sigma^2 estimated as 129942:  log likelihood = -1229.71,  aic = 2469.42
```

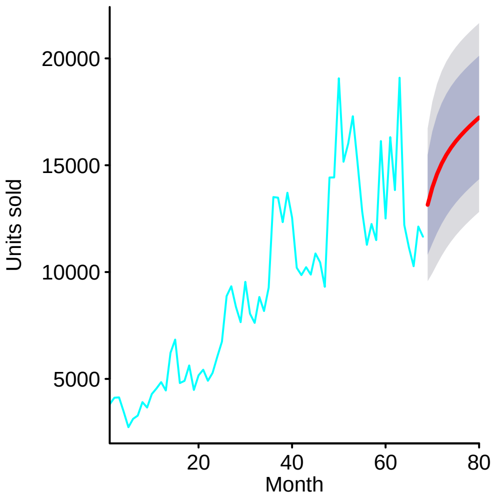


Figure 11.67: Monthly sales of spreadsheets in the UK, starting January 1987, with 12-months of sales predictions (shaded light blue are 80% confidence intervals, grey shaded 95%). Data from Givon et al.[676] Github–Local

The sar1 is the seasonal AR coefficient and sma1 the seasonal MA coefficient.

The output from auto.arima is a suggested model. In this case the ar and sar coefficients are pulling in opposite directions, and the standard error for the ma1 coefficient is very high. Removing the MA component produces a model (second call to arima above), where the coefficients are not almost cancelling each other out; the model is (24 is the seasonal period):

$$x_t = -0.4x_{t-1} - 0.1x_{t-2} + 0.9x_{t-24\times1} - 0.5w_{t-24\times1}$$

What happened 24 hours ago contributes more to the predictor, than what happened in the previous hour or two.

Predictions made using a fitted ARMA model: A fitted ARIMA model can be used to predict what may occur after a measurement at time t; the relatively large noise component present in some ARMA models means that the confidence bounds of the predicted values may quickly become very wide. The R's system supports a predict function that accepts models fitted by the arima function; the Arima and forecast functions, from the forecast package, support more options for fitting and forecasting of time-series data. In the following code, the include.drift=TRUE option specifies that trend information be included in the model; for this data, the increasing volume of sales generates an increasing trend:

```
library("forecast")

Ar_mod=Arima(data$Spreadsheets, order=c(1, 0, 1), include.drift=TRUE)

f_pred=forecast(Ar_mod, h=10)
plot(f_pred, col=point_col, main="", xaxs="i",
        xlab="Month", ylab="Monthly sales\n")
```

A study by Givon, Mahajan and Muller[676] investigated UK sales of PC's, wordprocessors and spreadsheets. Figure 11.67 shows monthly sales of spreadsheets, and based in an arima model, 12-months of predicted sales; shaded areas are 80% and 95% confidence intervals.

11.10.3 Non-constant variance

Time-series data containing rapid changes in variance is said to be *volatile*; correlated variance is common during periods of volatility (a time series is *heteroskedastic* if the change in variance is regular, and *conditionally heteroskedastic* if the change is irregular). Techniques for building an autoregressive model, for the variance, include *autoregressive conditional heteroskedastic* (ARCH) and *generalised ARCH* (GARCH) models.

An increase in frequency of commits leading up to a major new release is an example of behavior that can cause a change of variance in a time series.

The autocorrelation of a time-series may not show any correlation, but if its variance changes the square of the zero adjusted values will have a pattern of decreasing correlation in its ACF, as seen in figure 11.68; the code is:

```
acf(t_series)
acf((t_series-mean(t_series))^2)  # Check for changing variance
```

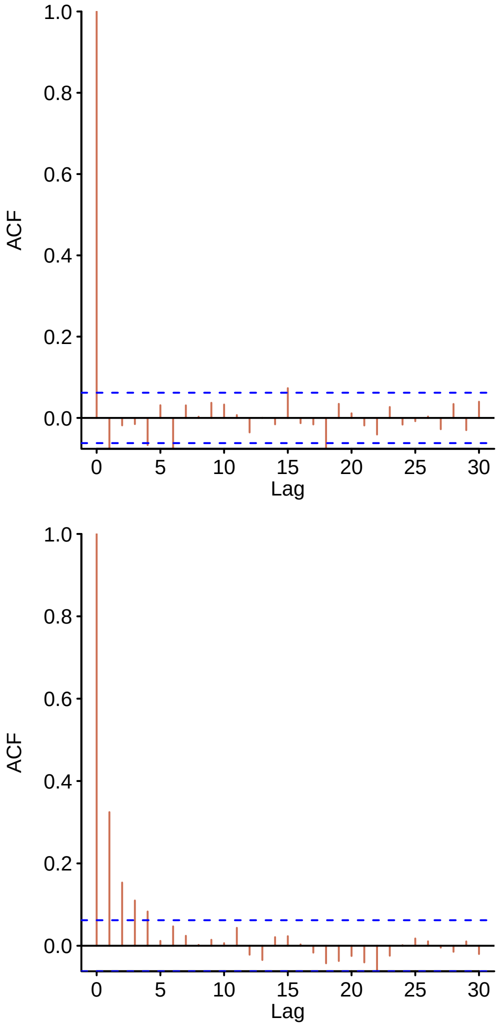


Figure 11.68: Time series whose values are uncorrelated (upper), but whose squared values are correlated (lower); see code for generation process. Github–Local

The rugarch package supports the fitting of GARCH models; see Github–time-series/splc-2010-fm.R.

A study by Lotufo, She, Berger, Czarnecki and Wąsowski[1150] investigated the evolution of the Linux variability model, through the lens of commits to Kconfig files. Figure 11.69

shows the number of commits per week made to the Linux kernel source and its associated Kconfig files. The commit bursts occur immediately prior to new releases.

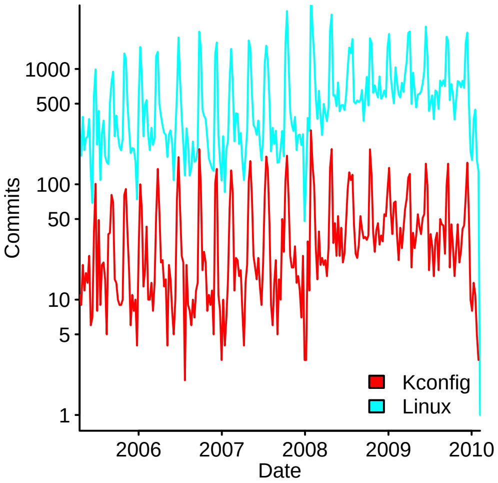


Figure 11.69: The number of commits per week to Linux kernel source and its Kconfig files. Data kindly provided by Lotufo.[1150] Github–Local

11.10.4 Smoothing and filtering

Smoothing a time-series can make it easier to visually identify larger scale patterns, and also provides a simple approximation to predicting immediate future values. Even when data does not contain a systematic trend, or seasonal effects (perhaps because they have been removed), it may still be possible to make a useful estimate of immediate future values based on immediate past values.

Smoothing using the *exponentially weighted moving average* (EWMA; also known as *exponential moving average*, EMA) uses the formula:

$EMA_t = \phi x_t + (1-\phi)EMA_{t-1}$, where: ϕ determines the amount of smoothing.

The *exponential moving standard deviation* (EMS) is given by:

$$EMS_t = \sqrt{\phi EMS_{t-1}^2 + (1-\phi)(x_t - EMA_t)^2}$$

EMA and EMS can be used to detect when a real-time data stream trends outside pre-specified bounds.

Holt-Winters smoothing is a generalization of exponential smoothing, that uses three parameters: estimated level, slope and seasonality; the HoltWinters function can be used to both estimate and apply these parameters.

The filter function can be used to apply AR and MA filters to a time series.

11.10.5 Spectral analysis

A series of measurements in the time domain can be transformed into a sequence in the frequency domain; see fig 1.12.

The spectrum function estimates the spectral density of a vector, which is assumed to be a time-series (the default behavior is to call the spec.pgram function). The spec.arma function takes a specification of an ARMA model, and returns its power spectrum, i.e., behaves like a call to spectrum when passed a time series that follows this model.

A stationary time-series does not contain components at specific frequencies, but can be described in terms of an average frequency composition.

11.10.6 Relationships between time series

Time series analysis can be used to find relationships between multiple time series, where each time series comes from measuring separate variables associated with some evolving process.

The simplest technique is cross-correlation, the correlation, at various lags, between two stationary time-series. Figure 11.70 shows the cross-correlation between the number of source lines added/deleted, per week, to the glibc library. In calls to the ccf function, the first argument is the one which is shifted, while the second is fixed. In the following call:

```
ccf(lines_added, lines_deleted, col=point_col, xlab="Weeks")
```

the plot shows correlation spikes, above the confidence bounds, occurring between the sequence pairs lines_added$_{t+2}$/lines_deleted$_t$ and lines_added$_{t+8}$/lines_deleted$_t$ (i.e., changes involving lines_deleted is correlated with changes to lines_added two and 10 weeks later; a positive lag means the first argument follows the second, a negative lag that it leads the second); there are small spikes at: lines_added$_{t-8}$/lines_deleted$_t$ and lines_added$_{t-13}$/lines_deleted$_t$. Your author has no explanation for this correlation.

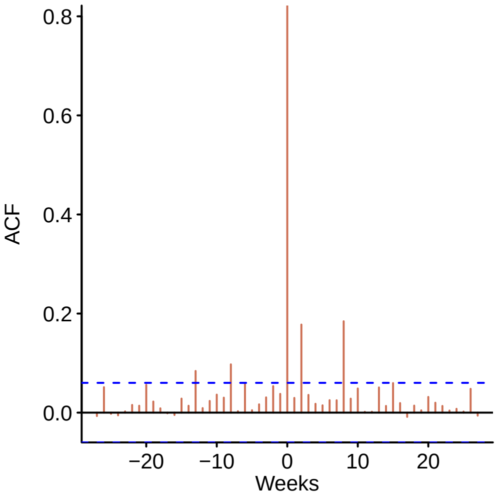


Figure 11.70: Cross-correlation of source lines added/deleted per week to the glibc library. Data from González-Barahona.[696] Github–Local

Techniques are available for building models of the relationship between two time series.

After making a commit to the Linux kernel, it may be discovered that an associated Kconfig file needs to be updated, i.e., the pattern of commits to Kconfig files will lag that of the commits of Linux source. Figure 11.71 shows the first six months of the two time series in figure 11.69, with the number of Kconfig commits shifted up to align with the kernel commits. The Kconfig commits often lag behind kernel commits.

The following calls to the `lags.select` function report a lag of 2-weeks for binned weekly data, and 7-9 days for daily data (which contains many zeroes):

```
library("tsDyn")

# Oldest comes first, and they need to be the same length
# By week
lags.select(cbind(head(log(linux_week$freq), -1), log(kconfig_week$freq)))
# By day
lags.select(cbind(head(log(linux_day$freq+1e-1), -2),log(kconfig_day$freq+1e-1)))
```

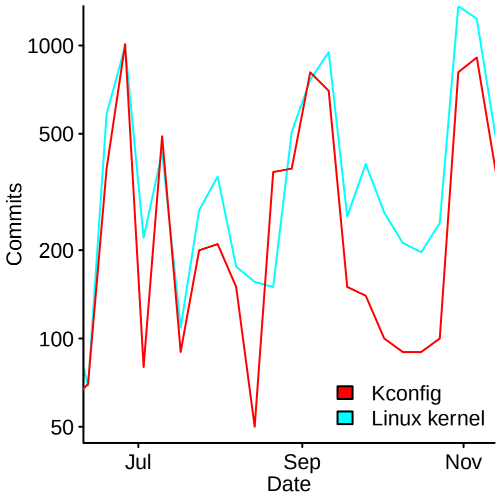


Figure 11.71: The number of commits per week to Linux kernel source and its Kconfig files, during the last half of 2005. Data kindly provided by Lotufo.[1150] Github–Local

What are the interdependencies between Linux source commits and Kconfig commits? The following code fits a Vector Autoregression (VAR) model (see Github–time-series/kconfig-evol.R):

```
library("tsDyn")

# Oldest comes first, using lag returned by lags.select
day_mod=lineVar(cbind(head(log(linux_day$freq+1e-1), -2),
                                         log(kconfig_day$freq+1e-1)),
                     lag=9)
```

the fitted equations for, *log*, Linux and Kconfig daily commits, are (the error terms have been omitted for brevity):

$$
\begin{aligned}
L\_commits_t =& 1.6 + 0.2L\_commits_{t-1} + 0.07K\_commits_{t-1} + 0.08K\_commits_{t-2} \\
&+ 0.09L\_commits_{t-3} + 0.1L\_commits_{t-6} + 0.1L\_commits_{t-7} - 0.09L\_commits_{t-9} \\
K\_commits_t =& -1.1 + 0.3L\_commits_{t-1} + 0.09K\_commits_{t-1} + 0.09K\_commits_{t-2} \\
&+ 0.06L\_commits_{t-3} + 0.2L\_commits_{t-6} + 0.09K\_commits_{t-7} - 0.1L\_commits_{t-9}
\end{aligned}
$$

showing Kconfig commits having a small influence, over a few days, on Linux commits, and Linux commits having a larger and longer term impact on Kconfig commits, than even earlier Kconfig commits.

Other forms of relationship that may exist between two or more time-series include:

Alignment: A time series is a sequence of values, with each value being larger, smaller or equal to the value immediately before it. If two time series are generated by the same, or similar, process they may contain subsequences of values that share the same pattern of up, down and no-change. A non-time series application of this kind of subsequence matching is extracting word sequences that commonly appear in two or more documents.

Dynamic time warping (DTW) is a class of algorithms that compares two series of values by stretching or compressing one of them (treated as the reference series), so it resembles the other (treated as the query series). The `dtw` package contains functions to perform and support DTW alignment of two series.

A study by Herraiz[1785] investigated the evolution of various long-lived software systems, and measured the growth of NetBSD and FreeBSD (in lines of code). These two operating systems started from the same base, continue to share developers (see fig 9.22) and code continues to be ported between them. Figure 11.72 shows the alignment, found by a call to `dtw`, between the weekly measurements of the lines of code in each OS (for the first 100 weeks of their development).

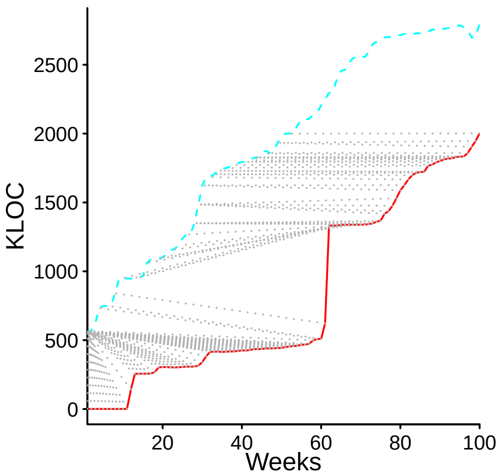


Figure 11.72: Visualization of alignment between lines of code, in NetBSD's (blue) and FreeBSD's (red) first 100 weeks. Data from Herraiz[1785] Github–Local

```
library("dtw")

bsd_align=dtw(freebsd_weeks, netbsd_weeks, keep=TRUE,
                   step=asymmetric, open.end=TRUE, open.begin=TRUE)
plot(bsd_align, type="twoway", offset=1, col=pal_col, xlab="Weeks")
```

Clustering: The pair-wise similarity of multiple time-series can be used as a clustering metric. Many techniques for measuring the distance between two time-series have been invented (at the time of writing, the `diss` function, in the `TSclust` package, supports 22 distance metrics).

A study by Powell[1497] investigated task effort allocation in a development project at Rolls-Royce. Figure 11.73 shows effort (in person hours) spent on eight major tasks (lower plot, from the bottom up: s/w requirements, top-level design, coding, low level test, requirement test, system acceptance test, management and holiday/non-project), and a hierarchical clustering of each task by its effort time series, with pair-wise distance between time series calculated using correlation (upper) and Euclidean (middle) metrics.

```
library("TSclust")

eff_dist=diss(t(all_effort), METHOD="COR")
plot(hclust(eff_dist), main="", sub="", xlab="", ylab="Correlation distance")
```

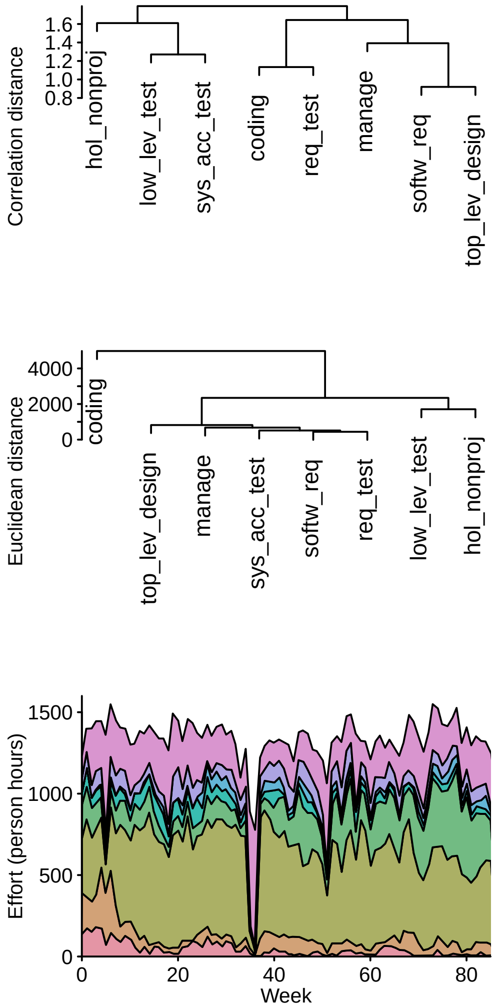


Figure 11.73: Effort distribution (person hours) over the eight main tasks of a development project at Rolls-Royce and a hierarchical clustering of each task effort time series based on pair-wise correlation and Euclidean distance metrics. Data extracted from Powell.[1497] Github–Local

11.10.7 Miscellaneous

Regression of time series data: Some of the issues involved in building regression models with serially correlated data are discussed in section 11.2.7.

Stochastic processes: Events involving future uncertainty may be modeled as a stochastic process; see section 3.2.4. The Sim.DiffProc package can be used to numerically solve stochastic differential equations of the Itô type;[226] section 3.2.4 discusses this topic in more detail.

The *Ornstein-Uhlenbeck process* is the continuous time version of an AR(1) model,[496] and the AR(1) process corresponding to equation 3.1 is:

$$x_t - x_{t-1} = \hat{x}(1-e^{-\eta}) + (e^{-\eta}-1)x_{t-1} + \varepsilon_t$$

Matrix profile[976] is an efficient new technique for finding motifs in time series. The tsmp package supports a variety of matrix profile related techniques; see Github–time-series/BSD-dtw.R.

11.11 Survival analysis

Survival analysis is the analysis of data where the response variable has the form of *time-to-event*. Historically this kind of model building has been used to compare the impact of different medical procedures, or drugs, on subject survival rate.

Survival analysis often deals with one kind of event, which causes a transition to a terminal state, e.g., there is returning from the dead. Competing risk models deal with the situation where one of several risk events can cause the transition to the final state. Multistate models handle the situation where some transitions are to states that are not final, i.e., an appropriate event can cause a transition to another state.

In some cases, the event of interest may not occur during the measurement period, in this case the measurement is said to be *censored*; for instance, when measuring the time interval between a function definition being written and the first time it is modified, the measurement data is said to be *right censored*, when one or more functions have not been modified over the time interval for which data is available.

Survival analysis makes greater use of available censored information, to produce estimates containing less error, than other forms of regression modeling; a linear regression model comparing mean time-to-event between groups would have to ignore censored data, while a logistic regression model, using 0/1 to indicate whether a subject survived or not, would again have to ignore censored data.

Possible outputs from survival analysis include:

- a survival function, $S(t)$, the probability of surviving a given amount of time. This can be used to estimate time-to-event for a group of subjects or compare time-to-event between subjects in two or more groups,
- a hazard function, $h(t)$, the hazard rate, that is, the probability of an entity surviving to time t experiencing an event in the next time interval, e.g., having survived 69 years 11 months before reading this sentence the probability that you die in the next month (the interval used to denote an instant is small compared to the time spans involved). The survival and hazard functions can be derived from each other:

  $$h(t) = \frac{f(t)}{S(t)}$$

  where: $f(t)$ is a probability density function, the probability of the event occurring at exactly t time units in the future, e.g., the probability of a baby born 70 years ago living long enough to read this sentence, but not before,

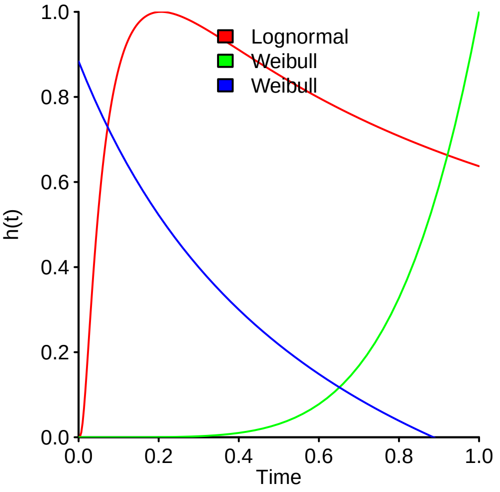


Figure 11.74: Two commonly used hazard functions; Weibull is monotonic (always increases, decreases or remains the same, depending on the equation coefficients), and Lognormal which can increase and then decrease. Github–Local

- a regression model specifying the impact of explanatory variables on time-to-event. This may be a non-parametric model, such as the Cox proportional hazard model, because parametric models can be very difficult to build.

Time-to-event is always positive and so has a skewed distribution (which means it cannot have a Normal distribution).

The `survival` package contains functions implementing the functionality needed to perform survival analysis.

11.11.1 Kinds of censoring

Ideally censoring is uninformative, i.e., the distribution of censoring times provides no information about the distribution of survival times. When a period of study is decided in advance, the censoring information is uninformative.

When censoring is not under experimenter control, it is said to occur at random. For instance, a subject may decide to stop taking part in a study because they are not happy with their performance.

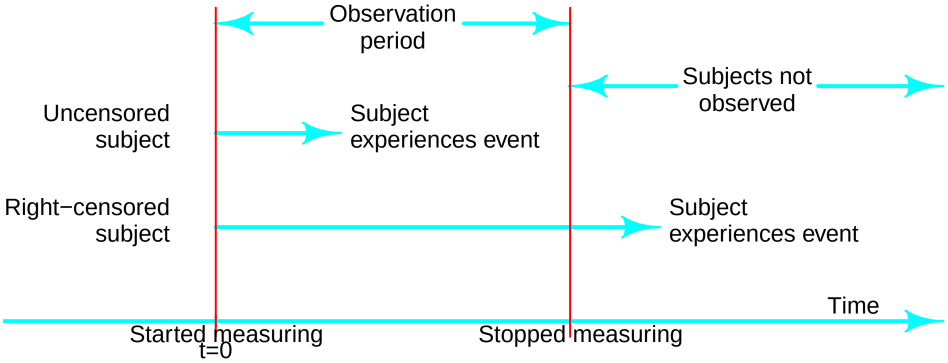


Figure 11.75: Observation period of study, with events inside and outside the study period. Github–Local

Kinds of censoring that can occur include:

- *left truncation*: subject not observed before t_0, experienced an event before that time and is not included in the study (the event may have been such that it rendered the subject unable to join the study, e.g., developer left the company),
- *left censored* (also *left truncated*): a subject included in the study is known to have had the event prior to time t, but with the exact time not being known,
- *right censored*: described at the start of the subsection,
- *interval censored*: when measurements are made at regular intervals, the exact time of an event is not known, only that it occurred between two measurement points,
- *non-detect*: the measurement process may fail to detect an event because the strength of the event is below the detection threshold. This kind of censoring is not covered here, see Helsel.[795]

11.11.1.1 Input data format

The `Surv` function creates a survival object from data, and the object it returns plays the role of the response variable in formula passed to model building functions. The required data format depends on the kind of censoring and presence of time dependencies. The following is an example of the basic information required:

```
id,start_time,end_time,failure_status,explanatory_v1,explanatory_v2
```

where: `id` is a unique identifier denoting each subject (only needed when information on the same subject occurs on multiple lines), `start_time/end_time` the starting time (or date) of measurement, and the end time (either when the event occurred, the end of the study or the last recorded time of a subject who was not seen again) and `failure_status` one of two values specifying whether an event occurred or not; followed by an optional list of explanatory variables.

The time of interest is the difference between the start/end time, and the data may contain just this value.

11.11.2 Survival curve

The *Kaplan-Meier* curve is a descriptive statistic of time-to-event measurements; it shows the percentage of subjects who have not experienced an event up to a point in time, along with an optional confidence interval (see figure 11.76)

The median is preferred over mean as the measure of central tendency for survival data, because the mean underestimates the true value when samples contain censored data. The median is measured as the point where the Kaplan-Meier curve falls before 0.5 and printing the model returned by survfit gives this value along with its 95% confidence intervals.

A study by Businge[279] investigated the number of releases of Eclipse third-party plug-ins (ETP) between 2003 and 2010; the history of each ETP was traced from the year of its first release and any releases in subsequent years were noted.

The Eclipse framework includes a published list of officially recognised APIs, but each Eclipse SDK release also includes support for APIs considered to be for internal use, i.e., not to be used by applications. The status difference between official/internal APIs is that internal APIs can be changed without notice, while the official APIs are intended to have some degree of permanence (they may change on major releases but are not intended to change on minor releases; starting in 2004 all yearly releases were minor).

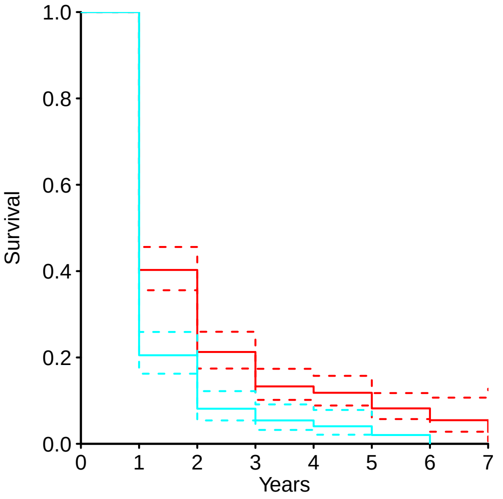


Figure 11.76: The Kaplan-Meier curve for survivability of new releases: (blue) ETPs using only official APIs, (blue) ETPs calling internal APIs (red); dotted lines are 95% confidence intervals. Data from Businge.[279] Github–Local

At some point there are no new releases of an ETP in a year and this cessation of new releases could be regarded as the *death* of development of the ETP (some ETP development died for one year only to be resurrected the following year; for simplicity the small number of such recurring events are ignored).

For this analysis ETP yearly release counts are divided into two groups, those that only made use of official APIs, and those that made use of one or more internal APIs; table 11.5 shows the number of ETPs using only the official API.

	2003	2004	2005	2006	2007	2008	2009	2010
2003	35	10	3	1	1	2	0	0
2004		33	4	4	2	2	0	0
2005			41	10	4	3	1	1
2006				61	7	1	0	2
2007					37	12	4	6
2008						38	7	2
2009							25	3
2010								16

Table 11.5: Total number of distinct ETPs released in a year; left column lists year of first release and releases in subsequent years. Data from Businge.[279]

Figure 11.76 shows the Kaplan-Meier curve for ETPs using only official APIs (blue) and ETPs that use internal APIs (red); the dotted lines are 95% confidence intervals. The following is the essential code (calling the Surv function to create a survival object, containing time and censored information on each subject, is the first step in most survival analysis using R):

```
library("survival")

api_surv=Surv(all_API$year_end-all_API$year_start,
                  event=(all_API$survived == 0), type="right")
api_mod=survfit(api_surv ~ all_API$API)
plot(api_mod, col=pal_col, conf.int=TRUE,  xlim=c(0,7), xlab="Years")
```

The summary function can be used to obtain values of the survival curve at each time measurement point.

Comparing two survival curves: Are the two survival curves statistically different? The survdiff function can be used to answer this question. The p-value returned by the call (bottom right) shows that the two survival curves are very unlikely to be the same: Github–Local

```
Call:
survdiff(formula = Surv(year_end - year_start, event = (survived ==
    0), type = "right") ~ API, data = all_API)
```

```
        N Observed Expected (O-E)^2/E (O-E)^2/V
API=0 381      334      372      3.83        29
API=1 289      260      222      6.41        29

 Chisq= 29  on 1 degrees of freedom, p= 7e-08
```

By default, `survdiff` performs a *log-rank test*, which gives equal weight to all events. Passing the argument `rho=1` causes greater weight to be given to earlier events, while the argument `rho=-11` gives greater weight to later events. The hazard function is returned by `survfit` functions when is it passed the argument `type="fh"`.

Why, on average, do new releases of an ETP using internal APIs occur over a greater number of years? Is it because there are changes to the internal APIs that break the ETP, requiring the ETP to be updated to handle the change and a new version released, or is it because authors who use internal APIs are more committed to creating the best possible product and so continue to refine their ETP over more years?

Perhaps, suspecting that changes to the SDK were a significant factor, Businge[279] investigated the source compatibility of ETPs with the Eclipse SDK across releases 1.0 to 3.7 (i.e., releases in every year from 2001 to 2011). Every ETP was built using each of these 11 SDK releases (yes, even SDKs created before an ETP was first released). To allow easy comparison with the ETP analysis above, the following analysis only considers SDK builds released after an ETP was first made available.

Figure 11.77 shows the survival of ETPs' ability to build under Eclipse SDKs released in each successive year. ETPs using internal APIs (red) are much more likely to fail to build (precompiled plug-ins may still function, if they don't call any changed internal API) when a new Eclipse SDK is released, compared to ETPs using only the official APIs (blue).

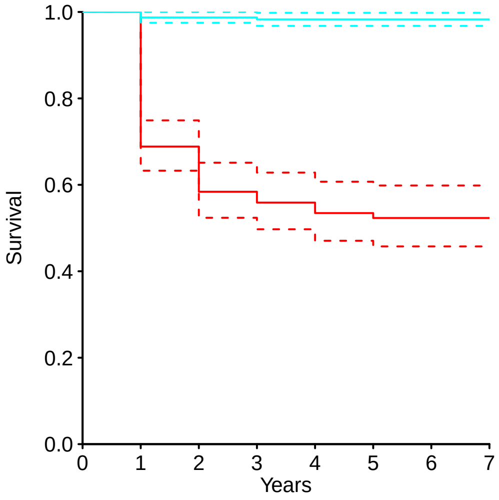


Figure 11.77: The Kaplan-Meier curve for survivability of ETPs ability to be built using SDK released in subsequent years: (blue) ETPs using only official APIs, (red) ETPs calling internal APIs; dotted lines are 95% confidence intervals. Data from Businge.[279] Github–Local

This analysis suggests that developers using internal APIs in their ETP, are more likely to be forced to release an update, if they want their ETP to continue to function with later releases. However, this data does not address the possibility that developers who make use of internal APIs are more committed to creating the best possible product.

11.11.3 Regression modeling

Survival data implicitly contains information that is not present in other forms of regression modeling: the probability of an event occurring at a given time, i.e., a hazard function. Estimating the appropriate hazard function for survival data requires knowing the coefficients of the explanatory variables in the regression model, while estimating the coefficients of the explanatory variables requires knowing the hazard function.

When building a model, R functions will attempt to fit the shape of the hazard function specified (by the analyst), but if this hazard function is incorrect, the returned model may be substantially incorrect. In practice, parametric models have been found to be very sensitive to the explanatory variables provided as input to the model fitting process.

There is no single statistic available for definitively selecting the best model (i.e., hazard function and appropriate explanatory variables).

The Cox proportional-hazards model does not require the specification of a hazard function, which breaks the circularity of needing to select regression coefficients for such a function and removes some of the dangers associated with use of an incorrect hazard function (the Cox modeling approach is not guaranteed to always build a reasonably accurate model). If there is any doubt about the appropriate parametric distribution, the Cox model is a safe choice.

While the Cox proportional hazards model has many advantages, a potentially big disadvantage is that without specifying a hazard function, it is not possible to make predictions outside the interval covered by the measurements.

The `flexsurv` package supports the fitting of complex parametric distributions, and the `censReg` package supports fitting regression models to censored data.

11.11.3.1 Cox proportional-hazards model

The Cox proportional-hazards regression model has been found to provide reasonably good estimates for the coefficients of the explanatory variables and hazard ratios (not absolute values, ratios) for a wide variety of data. The Cox model is popular because it is robust, and will closely approximate the correct parametric model. If the correct parametric model has a Weibull hazard function (whose shape parameter is unknown), the Cox model will give similar results to those obtained from this parametric model. If the parameters of the Weibull hazard function are known, a model built using them will outperform a Cox model.

The Cox likelihood (known as a *partial likelihood*) is based on the observed order of events, rather than the interval between them (so it only considers subjects' experiencing an event).

In the equation for the basic Cox model, time is not included as an explanatory variable, x_{ki}, i.e., the variables cannot be time dependent. The equation is:

$$h_i(t) = h_0(t)e^{\beta_1 x_{1i} + \cdots + \beta_k x_{ki}}$$

where: $h_i(t)$ is the hazard function for subject i at time t, $h_0(t)$ is a baseline hazard function, the contents of the exponent expression are explanatory variables and their regression coefficients (β_0 is included as part of the baseline hazard).

This equation can be written as a log ratio of the hazard functions:

$$\log \frac{h_i(t)}{h_0(t)} = \beta_1 x_{1i} + \cdots + \beta_k x_{ki}$$

or, as a hazard ratio for two subjects, i and j (where, $h_0(t)$, the baseline hazard function cancels out):

$$\frac{h_i(t)}{h_j(t)} = e^{\beta_1(x_{1i} - x_{1j}) + \cdots + \beta_k(x_{ki} - x_{kj})}$$

In this proportional hazards model, the effect of each explanatory variable is multiplicative on the hazard function. In accelerated failure time (AFT) models the multiplicative effect is on the survival function.

The `coxph` function, in the `survival` package, builds Cox proportional-hazard models; the basic usage follows the pattern used by `glm`, with the object returned by `Surv` playing the role of the response variable. For example:

```
p_mod=coxph(Surv(patch_days, !is_censored) ~ log(cvss_score)+opensource,
                                                              data=ISR_disc)
```

The `cox.zph` function can be used to check the assumption that the explanatory variables are not time dependent (at least during the measurement period).

If two or more events occur at the same time the associated data is said to be *tied*. The default value of the option `ties="efron"`, can handle some tied data, but if many events occur at the same time (e.g., the ETP data in table 11.5), calls to `coxph` might need to use `ties="exact"`.

The techniques for formula specification and refinement used with `glm` can also be applied to models created with `coxph`, e.g., starting with a complicated model and using `stepAIC` to simplify it.

A study by Arora, Krishnan, Telang and Yang[75] investigated the time taken by vendors to release patches, to fix vulnerabilities reported in their product; explanatory variables included information about the software vendor, whether the vendor was privately notified about the vulnerability, or the vendor first found out about it through a public disclosure.

The following is the `summary` output from a model fitted by `coxph` to the data for public disclosure vulnerabilities: Github–Local

```
Call:
coxph(formula = Surv(patch_days, !is_censored) ~ log(cvss_score) +
    opensource + y2003 + smallvendor + small_loge + log(cvss_score):y2002 +
    y2002:smallvendor + y2003:smallvendor, data = ISR_np)

  n= 945, number of events= 824

                          coef exp(coef) se(coef)      z Pr(>|z|)
log(cvss_score)        0.23283   1.26217  0.08570  2.717  0.00659 **
opensource             0.42235   1.52555  0.09167  4.607 4.08e-06 ***
y2003                  0.83643   2.30811  0.10459  7.997 1.27e-15 ***
smallvendor           -0.40940   0.66405  0.17331 -2.362  0.01816 *
```

```
small_loge             0.02926   1.02969  0.01346  2.173  0.02975 *
log(cvss_score):y2002  0.23048   1.25920  0.04961  4.646 3.39e-06 ***
smallvendor:y2002      0.59685   1.81638  0.19540  3.054  0.00226 **
y2003:smallvendor      0.58999   1.80396  0.22502  2.622  0.00874 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

                      exp(coef) exp(-coef) lower .95 upper .95
log(cvss_score)           1.262     0.7923    1.0670    1.4930
opensource                1.526     0.6555    1.2747    1.8258
y2003                     2.308     0.4333    1.8803    2.8332
smallvendor               0.664     1.5059    0.4728    0.9326
small_loge                1.030     0.9712    1.0029    1.0572
log(cvss_score):y2002     1.259     0.7942    1.1425    1.3878
smallvendor:y2002         1.816     0.5505    1.2384    2.6640
y2003:smallvendor         1.804     0.5543    1.1606    2.8039

Concordance= 0.647  (se = 0.011 )
Likelihood ratio test= 199  on 8 df,   p=<2e-16
Wald test            = 184.9  on 8 df,   p=<2e-16
Score (logrank) test = 198.3  on 8 df,   p=<2e-16
```

The first half of the output is similar to the `summary` output produced by a model fitted using `glm`. The table of numbers in the middle are 95% confidence intervals, which are printed by default. The bottom section of the output lists the R-squared of the fit[xxxi] (0.19 in this case, showing that only a small amount of the variance in the data is described by the model), and p-values for various tests of the null hypothesis that the coefficients are zero (abbreviated to a single letter, `p`).

The explanatory variable coefficients are proportions, not absolute values (the Cox model is a proportional-hazards model). The coefficients specify the expected impact of the respective explanatory variable, when the values of all the other variables are kept constant. The explanatory variables cannot be used to independently calculate response variable values, they can only be used to predict the change in a known value of the response variable (i.e., the value of the response variable known to occur for specific values of the explanatory variables).

Taking the values in the `log(cvss_score)` row as an example, the value 1.26217 appears in its `exp(coef)` column; what impact will a ±1 change in the value of $\log(cvss\_score)$ have on the response variable (i.e., time taken to produce a patch)? The percentage change in the response variable is: $\pm(1.2621-1)\times 100 \rightarrow \pm 26.21\%$; a value of less than one, in the `exp(coef)` column, reverses the sign of the percentage change, e.g., an increase in the value of the explanatory variable is predicted to decrease the value of the response variable.

Model adequacy can be checked using Cox-Snell residuals, and influential observations searched for using *score residuals* (which specify how each regression coefficient would change if a particular observation was removed; see Github–survival/vulnerabilities/patch-cph.R).

Frailty of subjects: Unobserved differences in subject performance may result in some variation in the hazard function they experience;[87] *frailty* is the term used to denote these random (multiplicative) changes in hazard function. Introducing a random effect, v_j, the frailty of group j that x_i belongs to, modifies the Cox model as follows:

$$h_i(t) = h_0(t)v_j e^{\beta_1 x_{1i}+\cdots+\beta_k x_{ki}}$$

The previous analysis of time-to-patch, implicitly assumes there is no difference between vendors, in their ability to respond and fix reported vulnerabilities. The `frailty` function can be included in a formula, to specify explanatory variables that identify particular groups of subjects sharing the same frailty.

```
fp_mod=coxph(Surv(patch_days, !is_censored) ~ log(cvss_score)+opensource
                                              +frailty(vendor), data=ISR_disc)
```

The `summary` output includes the information: `Variance of random effect=0.374` (see Github–survival/vulnerabilities/patch-frailty.R).

[xxxi] The value printed is the Cox & Snell pseudo R-squared, which can be less than one; the maximum possible value for the data is given in the `summary` output.

The v_j in the above equation is assumed to have a mean and variance that is calculated as part of the model building process (in this case it is 0.374). The main consequence of including frailty in a Cox model is to explicitly allocate some of the variance present in the data to a specific explanatory variable (the model coefficients of explanatory variables may also change).

The `frailtypack` package provides a wider range of frailty related options and functionality, than is available in the `survival` package. The `coxme` package supports the fitting of mixed-effects Cox models (frailty can be handled as a specific kind of random effect).

11.11.3.2 Time varying explanatory variables

The behavior of an explanatory variable may change over time. Options for handling this behavior includes, excluding all affected subjects from the analysis or using a technique that handles time dependent behavior.

The Arora et al study (discussed earlier) investigated the impact of public disclosure of vulnerabilities, on the time taken by vendors to release patches for their product. Possible event sequences were:

- vendor was privately notified about a vulnerability and some time later a simultaneous announcement of the vulnerability and a vendor patch was made (213 of 755 private notifications),
- vendor was privately notified about a vulnerability, but information about the vulnerability was later made public before a patch was available for release (the vendor's patch being released some time later in 542 of 755 private notifications); this is a time dependent change of a significant attribute.
- the vendor learned about a vulnerability when information about it was made public, and sometime later released a patch (945 cases),

If privately notified and public disclosure fix rates are compared using a Kaplan-Meier curve, any privately notified vulnerabilities that become public before a patch is available have to be treated as censored (ignoring them biases fix rates towards a lower value; see figure 11.78).

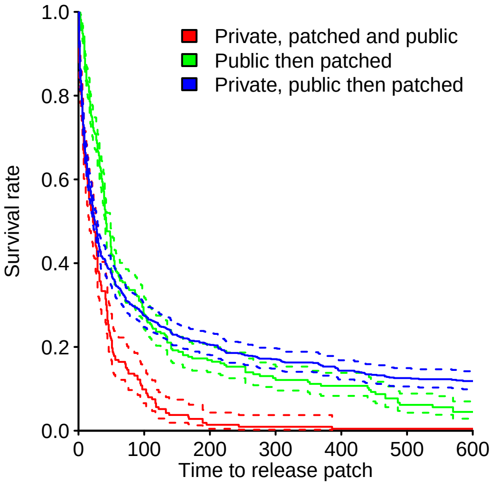


Figure 11.78: Kaplan-Meier curves for time-to-release a patch for a reported vulnerability, with private, public, and private then public notification. Data from Arora et al.[75] Github–Local

The Cox models discussed earlier, were fitted using vulnerability data where the vendor found out about the vulnerability via public disclosure.

Building a regression model using all the vulnerability data, requires handling time dependent explanatory variables; the data has to be restructured to make the time dependencies explicit. The time dependency, for this data, is a possible change of state, from the vulnerability not being public, to the information being public.

The original data looks something like the following:

```
notify,publish,patch,vendor,employee,os
2000-10-16,2000-11-18,2000-12-20,"abc",1000,unix
```

When vulnerability information is made public before a patch is released, extra information is involved. For this data, five columns are added: one to uniquely identify each vulnerability, the start/end dates of the interval during which the information was private or disclosed, a flag specifying private/disclosed, and a flag for whether an event (i.e., release of a patch) occurred in the interval. The first interval starts on the date the vendor was notified and ends on the date the vulnerability is made public, a second interval occurs for vulnerabilities that change state from private to disclosed before a patch is available and starts on the date of disclosure and ends on the date a patch became available, as follows:

```
id,start,end,priv_di,notify,publish,patch,event,vendor,os
1,2000-10-16,2000-11-17,1,2000-10-16,2000-11-18,2000-12-20,0,"abc",unix
1,2000-11-18,2000-12-20,0,2000-10-16,2000-11-18,2000-12-20,1,"abc",unix
```

Treating `priv_di` as an explanatory variable (1 for private disclosure to vendor and zero for public disclosure), enables the impact of disclosure on patch time to be included in a model.

When all the measurement data needs to be split on the same date, the `survSplit` function can be used to create the necessary rows, otherwise (as in this case) specific data mangling code has to be written.

The call to coxph, or survreg, has to include the term cluster(id), which ties together (by vulnerability id in this case) the rows associated with the same subject. The call to coxph looks something like the following:

```
td_mod=coxph(Surv(patch_days, !is_censored) ~ priv_di*cvss_score
                                               +cluster(id), data=ISR_split)
```

It is not possible for both cluster and frailty to appear in the same formula (cluster is based on GEE model building, while frailty is based on mixed-effects model building).

The summary output for the time dependent model is: Github–Local

```
Call:
coxph(formula = Surv(patch_days, !is_censored) ~ priv_di + cvss_score +
    y2 + small_loge + priv_di:cvss_score + priv_di:c_o + priv_di:dis_by_s +
    priv_di:os + priv_di:y2 + priv_di:smallvendor + priv_di:small_loge +
    cvss_score:c_o + cvss_score:dis_by_s + cvss_score:s_app +
    c_o:opensource + dis_by_s:opensource + os:s_app + y2:s_app,
    data = ISR_split, cluster = ID)

  n= 2242, number of events= 2081

                         coef exp(coef)  se(coef) robust se      z Pr(>|z|)
priv_di              2.798750 16.424106  0.216150  0.209360 13.368  < 2e-16 ***
cvss_score           0.153926  1.166404  0.016806  0.017733  8.680  < 2e-16 ***
y2                   0.277421  1.319722  0.044042  0.044590  6.222 4.92e-10 ***
small_loge           0.037114  1.037811  0.007262  0.008817  4.210 2.56e-05 ***
priv_di:cvss_score  -0.114788  0.891555  0.017327  0.016795 -6.835 8.22e-12 ***
priv_di:c_o          0.644347  1.904743  0.228463  0.211989  3.040 0.002369 **
priv_di:dis_by_s     0.475405  1.608665  0.116261  0.106601  4.460 8.21e-06 ***
priv_di:os          -0.331847  0.717597  0.098936  0.086976 -3.815 0.000136 ***
priv_di:y2          -0.614162  0.541094  0.063296  0.061954 -9.913  < 2e-16 ***
priv_di:smallvendor -0.440845  0.643492  0.138900  0.099310 -4.439 9.03e-06 ***
priv_di:small_loge  -0.082120  0.921161  0.016589  0.014449 -5.683 1.32e-08 ***
cvss_score:c_o      -0.060084  0.941685  0.012861  0.011990 -5.011 5.42e-07 ***
cvss_score:dis_by_s -0.061114  0.940716  0.008798  0.011002 -5.555 2.78e-08 ***
cvss_score:s_app    -0.096771  0.907764  0.014972  0.014853 -6.515 7.27e-11 ***
c_o:opensource       0.443978  1.558896  0.137952  0.118459  3.748 0.000178 ***
dis_by_s:opensource  0.414151  1.513086  0.091161  0.102359  4.046 5.21e-05 ***
os:s_app             0.815803  2.260991  0.077450  0.093536  8.722  < 2e-16 ***
y2:s_app             0.291007  1.337774  0.047420  0.045599  6.382 1.75e-10 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Concordance= 0.654  (se = 0.007 )
Likelihood ratio test= 590.1  on 18 df,   p=<2e-16
Wald test            = 376.9  on 18 df,   p=<2e-16
Score (logrank) test = 586.8  on 18 df,   p=<2e-16,   Robust = 454.6  p=<2e-16

  (Note: the likelihood ratio and score tests assume independence of
     observations within a cluster, the Wald and robust score tests do not).
```

There are two parts to the contribution made by priv_di; as a standalone variable it has a large impact, but its interactions with other variables create a large impact in the opposite direction (the model building process tries to minimise its error metric, not make it easy for analysts to understand what is going on).

The component of the fitted equation of interest is:

$$e^{priv\_di(2.8-0.11cvvs\_scorei+0.64c\_o+0.48dis\_by\_s-0.33os-0.61y2-0.44smallvendor-0.08small\_loge)}$$

where: *priv_di* is 0/1, *cvvs_score* varies between 1.9 and 10 (mean 7), *c_o* 0/1 NA other[xxxii] (mean 0.13), *dis_by_s* 0/1 disclosed by SecurityFocus (mean 0.38), *os* 0/1 vulnerability in O/S (mean 0.26), *y2* years since 2000 (mean 1.9), *smallvendor* 0/1 small vendor flag (mean 0.25) and *small_loge* zero for small vendors, otherwise the log of number of employees (mean 5).

Applying some hand waving to average away variables:

[xxxii] No information is available on what this variable represents.

$$e^{priv\_di \times (2.8 - 0.11 \cdot 7 + 0.64 \cdot 0.13 + 0.48 \cdot 0.38 - 0.33 \cdot 0.26 - 0.61 \cdot 1.9 - 0.44 \cdot 0.25 - 0.08 \cdot 5)}$$

$$\rightarrow e^{priv\_di \times (2.8 - 0.77 + 0.08 + 0.18 - 0.09 - 1.2 - 0.11 - 0.4)} \rightarrow e^{0.49 priv\_di}$$

produces a (hand waved mean) percentage increase of $(e^{0.49} - 1) \times 100 \rightarrow 63\%$, when `priv_di` changes from zero to one. The percentage change for patches, for vulnerabilities with a low `cvvs_score` is around 90% and for a high `cvvs_score` around 13% (i.e., the patch time of vulnerabilities assigned a low priority improves a lot when they are publically disclosed, but patch time for those assigned a high priority is only slightly affected).

The process of calculating the 95% confidence bounds, based on the values in the `summary` output, is fiddly and left to the reader.

Time varying changes may occur, but the information needed to model these changes may not be available.

A study by Lunesu[1159] investigated the maintenance activities of a large software company; between 2005 and 2010 there were 5,854 issues. The response time for an issue (i.e., the time between an issue being opened to it being closed) depends on the rate issues are reported, and the resources available to handle issues.

Extra resources were added to handle the growing number of issues (after around 400 days), and the number of issues reported decreased after around 800 days (it is not known whether this resulted in issue handling resources being decreased, or the same level of resources applied to fewer issues). The level of resources used to resolve issues cannot be included in a model, because this information is not available.

Figure 11.79 shows the cumulative number of issues reported and closed, over time (upper); the lower plot shows the survival curve for issues reported in the first 400 days, reported between 400 and 800 days and reported after 800 days. As expected, the extra resources added after 400 days reduced open issue survival times, but the reduction in reported issues after 800 days does not appear to have had much impact on survival rates (perhaps because it is easier to move existing staff to other work, than to add new staff).

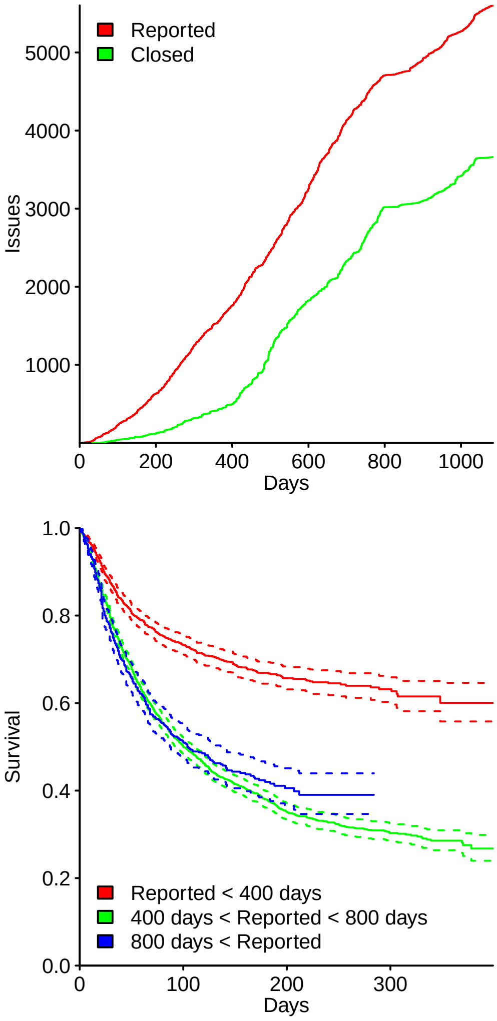


Figure 11.79: Cumulative number of issues reported and closed, and issue survival curves for three intervals. Data from Lunesu.[1159] Github–Local

11.11.4 Competing risks

When more than one possible kind of event can occur (i.e., there are multiple terminal states), a competing risk model might be used or each distinct event type might be analyzed separately from the other event types (data involving other events is flagged as censored at the time the other event occurs).

The Kaplan-Meier plot for a single event, in a competing risk context, may give a misleading impression of the actual situation for events that rarely occur. The *cumulative incidence curve* (CIC) is a commonly used alternative, which includes information on every event (when there is only one event, $CIC = 1 - KM$). CIC does not assume that competing risks are independent and estimates the marginal probability of an event.

The `cmprsk` package supports the modeling of competing risks.

A study by Di Penta, Cerulo and Aversano[485] investigated the history of mistakes in source code flagged by various static analysis tools. Newly written source code containing a flagged construct was tracked through subsequent versions. Possible competing events include the removal of the code containing the flagged construct and the flagged construct being modified such that it is no longer flagged (e.g., a bug fix).

Figure 11.80 shows the cumulative incidence curves (created by the `cuminc` function, in the `cmprsk` package) for problems reported in the Samba and Squid source by the splint static analysis tool.

```
library("cmprsk")

plot_cif=function(sys_str)
{
t=cuminc(rats$failtime, rats$type, cencode=0, subset=(rats$SYSTEM == sys_str))

plot(t, col=pal_col, cex=1.25,
        curvlab=c("was removed", "disappeared"),
        xlab="Snapshot", ylab="Proportion flagged issues 'dead'\n")

text(max(t[[1]]$time)/1.5, 0.9, sys_str, cex=1.5)
}

plot_cif("samba")
plot_cif("squid")
```

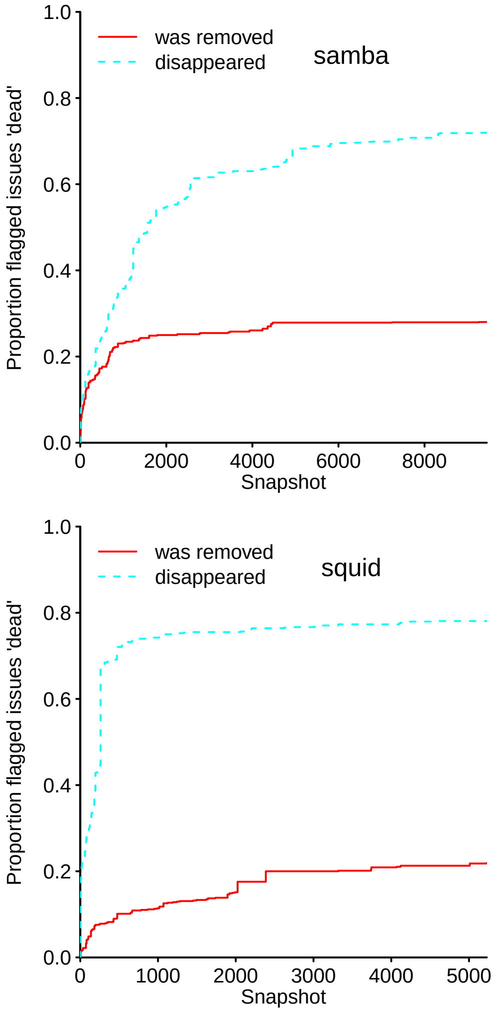


Figure 11.80: Cumulative incidence curves for problems reported by the splint tool in Samba and Squid (time is measured in number of snapshot releases). Data from Di Penta et al.[485] Github–Local

11.11.5 Multi-state models

Multistate models deal with time to event processes, where there are multiple events with potential changes of state between them.

The mstate package supports the building of multi-state Cox models and competing risk models, the msm package supports the building of multi-state Markov models.

A study by Goeminne and Mens[684] investigated the evolution in the use of four database frameworks in 2,818 Java projects. A project might start using, say, Spring, then add support for another framework or switch from using Spring to a different framework.

What is the probability of changes, over time, in the number of database frameworks, used by a project?

The following call to msm fits a multi-state Markov model for the number of database frameworks (dbs) used by projects (X) over time (date). The matrix Q is an initial estimate of the state transition probabilities of a project currently using *i* frameworks migrating to use *j* frameworks; any transition that cannot occur must contain zero in the corresponding non-diagonal element (specifying gen.inits=TRUE results in an approximate estimate being calculated internally; see Github–survival/icsme2015era.R).

```
library(msm)

Q=matrix(nrow=4, ncol=4,
                c(0,   0.1, 0.1, 0.01,
                  0.1, 0,   0.1, 0.01,
                  0.1, 0.1, 0,   0.1,
                  0.1, 0.1, 0.1, 0))

# Fit a multi-state Markov model
db_msm=msm(dbs ~ date, subject=X, data=uses,
                         qmatrix=Q, gen.inits=TRUE, exacttimes=TRUE)

# Extract the estimated transition probability matrix from the fitted
# model at time 365 (a year)
pmatrix.msm(db_msm, t=365)
# Estimate mean time spent in each transient state of the model
sojourn.msm(db_msm)
```

Table 11.6 shows the estimated likelihood of a project migrating from using *i* database frameworks to using *j* frameworks, within 365 days.

from	to 1	to 2	to 3	to 4
1	0.89	0.09	0.02	0.00
2	0.07	0.74	0.17	0.03
3	0.02	0.07	0.77	0.14
4	0.01	0.01	0.05	0.93

Table 11.6: Estimated likelihood that within 365 days, a project using *i* database frameworks will migrate to using *j* frameworks. Data kindly provided by Goeminne.[684]

11.12 Circular statistics

Some measurements are based on a circular scale, with values wrapping around to the minimum value when incremented past the maximum value, e.g., time of day, or months of the year. Circular statistics[1457] is the name given to the analysis of data measured using such a scale. Circular statistics has only become widely studied in the last 40 years or so, and techniques for handling operations that are well-established in other areas of statistics are still evolving. The circular package supports the analysis of data measured using a circular scale.

Differences between measurements based on circular and linear scales include:

- plotting uses a polar representation, rather than x/y-axis (the `circular` package includes support for `plot`, `lines`, `points` and `curve` functions),
- the mean has two components: mean direction ($\overline{\theta}$, an angle) and mean resultant length ($\overline{R}$; if this value is zero, the data has no mean), returned by the `mean` and `rho.circular` functions respectively (the `trigonometric.moment` function provides another way of obtaining this information). The `median.circular` function returns a median (multiple medians may exist, but only one is returned),
- the term variance, on its own, is ambiguous. The *circular variance*, V, is defined as $V = 1 - \overline{R}$ and varies between zero and one. Another measure is *angular variance* (returned by the `angular.variance` function), which varies between zero and two.

  The *circular standard deviation* is returned by the `sd.circular` function (it is not calculated by taking the square root of the variance; its formula is: $\sqrt{-2\log\overline{R}}$),
- the von Mises distribution plays a role similar to that filled by the Normal distribution on linear measurement scales.

The mean resultant length, $\overline{R}$, is a measure of how spread out data points are around the circle. If the points have a symmetric distribution $\overline{R}$ equals zero and if all the points are concentrated in one direction $\overline{R}$ equals one; for unimodal distributions the term *concentration* is applied to $\overline{R}$ to denote the extent to which measurements concentrate around the mean direction.

Figure 11.81 is a Rose diagram of the number of commits to Linux and FreeBSD for each 3-hour period of the days of the week (the same data is plotted using a linear scale in figure 5.6).

The `rose.diag` function plots Rose diagrams. By default, the area of each segment is proportional to the number of measurement points in the segment (the behavior used when plotting histograms).

```
library("circular")

# Map to a 360-degree circle
HoW=circular((360/hrs_per_week)*week_hr, units="degrees", rotation="clock")
rose.diag(HoW, bins=7*8, shrink=1.2, prop=5, axes=FALSE, col=col_str)
axis.circular(at=circular(day_angle, units="degrees", rotation="clock"),
        labels=c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"))

text(0.8, 1, repo_str, cex=1.4)
arrows.circular(mean(HoW), y=rho.circular(HoW), col=pal_col[2], lwd=3)
```

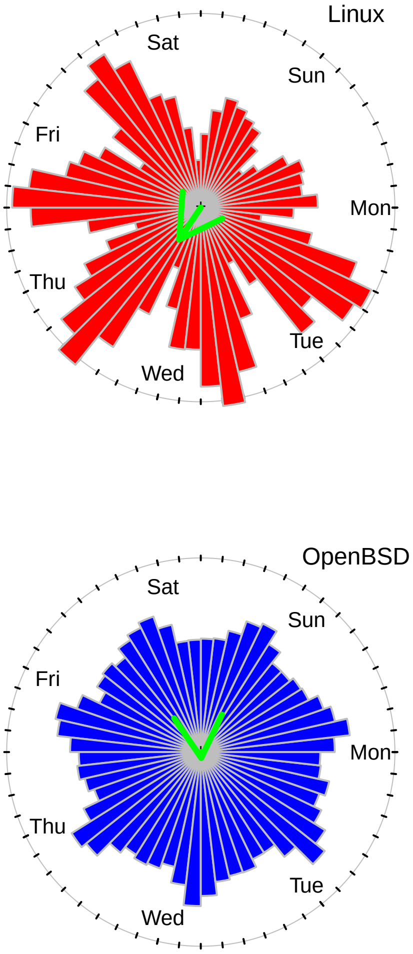


Figure 11.81: Rose diagram of number of commits in each 3-hour period of a day for Linux and FreeBSD. Data from Eyolfson et al.[558] Github–Local

The arrow at the center shows the direction of the mean, and the length of its shaft is its resultant length. Linux has fewer commits at weekends, compared to weekdays, and a mean direction near the middle of the week looks reasonable. The number of commits to FreeBSD does not seem to vary between days; the mean length is 0.03 (it almost does not have a mean), compared to Linux's mean length of 0.2.

If the measurement scale is very granular (e.g., measuring commit time once by day, rather than hour or minute), then $\overline{R}$ will be underestimated, and introduce errors in the calculation of the various location measures which use this value (see Github–statistics/circular/circle-bin.R). The correction to $\overline{R}$, for calculating circular standard deviation, when measuring in units of days rather minutes, is to multiply the calculated value of $\overline{R}$ by 1.034 (the correction for higher order moments involves much larger values).

11.12.1 Circular distributions

Circular distributions that are often encountered include the uniform distribution, wrapped Cauchy, wrapped von Mises and the Cartwright distributions.

Figure 11.82 shows differences in the shapes of three popular, symmetrical, single peak, wrapped circular distributions.[xxxiii] The *Jones-Pewsey distribution* includes them all, and others, as special cases.

Figure 11.83 shows asymmetric extended forms of some common circular distributions. The `circular` package does not include support for asymmetric distributions, but code is available in Pewsey et al.[1457]

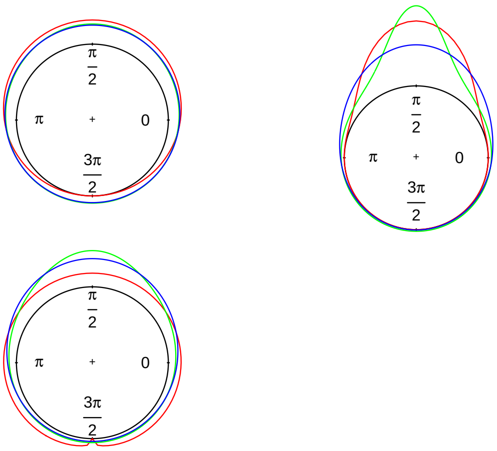


Figure 11.82: The Cartwright (red; `dcarthwrite`), wrapped Cauchy (green; `dwrappedcauchy`) and wrapped von Mises (blue; `dvonmises`) circular probability distributions for various values of their parameters. Github–Local

[xxxiii] The implementation of the Cartwright distribution, up to at least version 0.4.93 of the `circular` package, uses the spelling `carthwrite`.

The *discrete circular uniform distribution* consists of m points, equally spaced around a unit circle, with each point occurring with probability $\frac{1}{m}$.

The *continuous circular uniform distribution* treats all directions as being equally likely; the probability of point occurring between the angles ϕ and ψ is: $P(\phi < \theta < \psi) = \frac{\phi - \psi}{2\pi}$. The `circular` package supports the `dcircularuniform` and `rcircularuniform` functions, but not `p` and `q` forms.

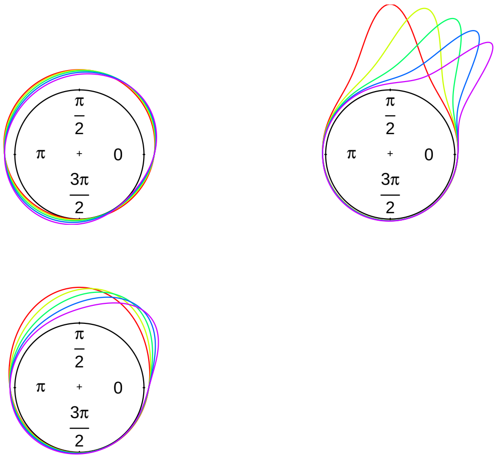


Figure 11.83: Asymmetric extended wrapped forms of the Cardioid (upper), von Mises (middle) and Cauchy (lower) probability distributions for various values of their parameters. Github–Local

The choice of which circular uniformity test to use, for measurements on a continuous scale, i.e., many possible measurement points around the circle, depends on how the data is thought to deviate from uniformity. The two uniformity deviation possibilities are:

- a single peak over some range of values, i.e., a unimodal distribution. In this case the Rayleigh test is the most powerful known test; available in the `rayleigh.test` function,
- multiple peaks in the distribution of values around the circle. There are three tests that are more powerful than the Rayleigh test, when the data distribution could be more complicated than a single peak, but no single one is superior to the others; support for these tests is available in the `kuiper.test`, `watson.test` and `rao.spacing.test` functions.

Unless there is a good reason to think that the measurements could have a single peak, one (or all) of the omnibus tests should be used.

Your author was once a member of a four-person compiler implementation team, all born in February; we all agreed that the best compiler implementers are born in February. An alternative, easier to test hypothesis, is that most compiler implementers are born in February. Your author ran a survey on his compiler oriented blog,[924] asking readers their birth month and whether they had spent more than four months working on a compiler project (132 responded, of which 82 had worked on a compiler).

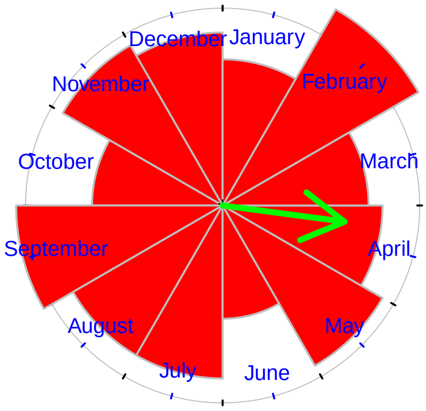


Figure 11.84: Number of readers of author's blog, whose birthday falls within a given month and who have worked on a compiler. Data from Jones.[924] Github–Local

Figure 11.84 shows that while February was the most common birth month, with 15% of implementers it is substantially below a majority (weighting by the number of days in a month, or the yearly percentage of births in each month, does not have much impact).

How likely is it that compiler implementer birthdays are uniformly distributed over the months? When the measurements have been grouped into a few bins, e.g., months of the year, a grouped data test has to be used; see Github–statistics/circular/grp-data-boot.R for examples using bootstrap. All tests for uniformity fail.

A study by Eyolfson, Tan and Lam[559] investigated the correlation between commit time and the likelihood of a fault being experienced because of a mistake in the commit code, for Linux and PostgreSQL. Figure 11.85 shows the number of non-fault commits (upper) and number of commits in which a fault was detected (lower), made in each hour of combined weekdays (the pattern of commits on weekdays differs from weekend days, and the following analysis is based on weekdays only).

What differences, if any, exist between the two sets of daily commit times and in particular are commits made at certain times of the day more likely to cause a fault experience?

- testing for a common mean direction: The `watson.williams.test` function assumes that both samples are drawn from a von Mises distribution; the *Watson large sample non-parametric test* does not even require the samples to share a common shape (see Github–statistics/circular/common-mean.R). When any sample has a size less than 25, a bootstrap version of these tests should be used. The daily commit times do not share a common mean direction (15.5 hours for fault commits and 16.2 hours for non-fault commits); the mean result lengths are 0.33 and 0.32 respectively,
- testing for a common concentration: Are the points concentrated around a common direction? The *Wallraff test* is not supported by the `circular` package, but is described in Pewsey et al[1457] (see Github–statistics/circular/common-concen.R). The two commit samples do not share a common concentration.

11.12.2 Fitting a regression model

When one or more variables are measured on a circular scale the technique used to build a regression model depends on whether the circular variable is an explanatory or response variable.

When the response variable is measured on a linear scale, existing techniques and functions can be used; there may be one or more circular or linear explanatory variables,

When the response variable is measured on a circular scale, the `lm.circular` function, in the `circular` package, can be used

11.12.2.1 Linear response with a circular explanatory variable

Circular explanatory variables can be modeling using periodic functions, and the regression modeling techniques discussed in earlier sections; sine and cosine functions can be combined to model any periodic function. As always, a model containing the fewest number of distinct parameters is desired.

The cosine function can be modified in various ways to change its shape, including:

$$y = \alpha + \beta\cos(\omega x + \phi)$$

and higher order harmonics can be added:

$$y = \alpha + \beta_1\cos(\omega x + \phi) + \beta_2\cos(2\omega x + \phi)\cdots$$

The shape of the peaks and troughs can be modified by adding a sine wave to the angular argument. In the following a positive λ sharpens the peaks and flattens the troughs while a negative λ has the opposite effect.

$$y = \alpha + \beta\cos(\omega x + \phi + \lambda\sin(\omega x + \phi))$$

A skewed period (which is what asymmetrical distributions have) can be modeled by adding a cosine wave to the angular argument (provided $-\pi/6 \leq \lambda \leq \pi/6$; outside this range it also affects other shape characteristics):

$$y = \alpha + \beta\cos(\omega x + \phi + \lambda\cos(\omega x + \phi))$$

These are all non-linear equations, which can be fitted using the `nls` function.

Figure 11.85 shows the number of commits to the Linux kernel, per hour; it is asymmetric, and the following code fits an extended cosine regression model (the `gam` values were estimated from the height of the cycle, and `omega` from fitting 24 hours into 2π radians):

```
basic_mod = nls(freq ~ gam0+gam1*cos(omega*hour-phi+nu*cos(omega*hour-phi)),
                 start=list(gam0=800, gam1=700, omega=0.3, phi=1, nu=0),
                 data=week_basic)
```

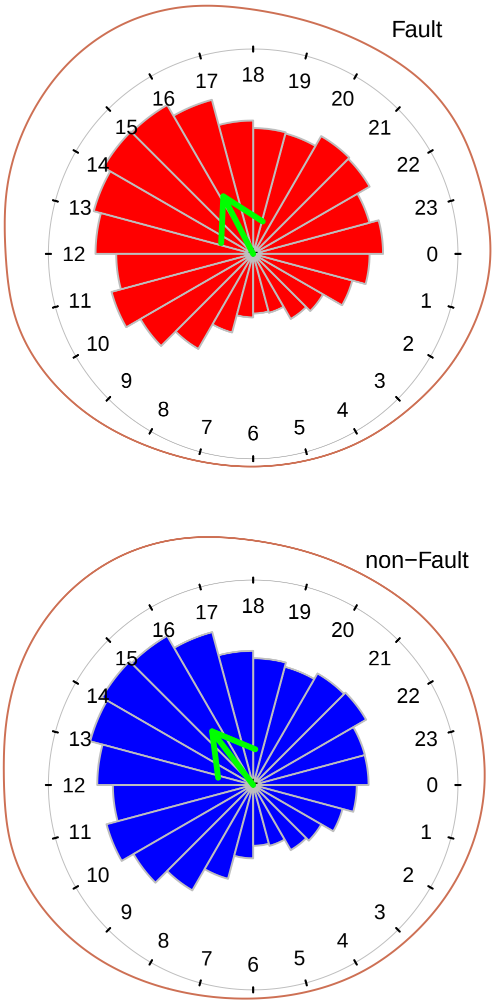


Figure 11.85: Number of commits (upper) and number of commits in which a fault was detected (lower) by hour of day of the commit, for Linux. Data from Eyolfson et al.[559] Github–Local

Figure 11.86 shows the number of non-fault related commits, and fault related commits, per hour for every week day; with fitted models.

Both fits handle the skewed period but not the sharp peak and flat trough. A sine contribution can be added to help handle this shape and improve the fit, the call to `nls` is below:

```
basic_2mod = nls(freq ~ gam0
                 +gam1*cos(omega*hour-phi+nu*cos(omega*hour-phi))
                 +gam2*cos(2*omega*hour-phi+nu*sin(omega*hour-phi)),
                     start=list(gam0=800, gam1=700, gam2=100,
                                omega=0.3, phi=1, nu=0),
             data=week_basic)
```

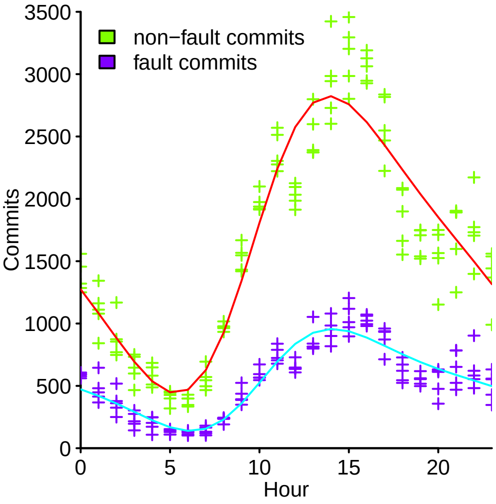


Figure 11.86: Number of non-fault related commits, and commits related to fixing a reported fault, per hour for weekdays, for linux; with fitted models. Data from Eyolfson et al.[559] Github–Local

Figure 11.87 overlays the fitted curve for non-fault and fault (red) commits over the non-fault hourly commits for each workday.

The `lm.circular` function supports circular response variables.

11.13 Compositional data

A study by Machiry, Tahiliani and Naik[1171] measured the performance of two application test generators, by comparing the number of lines of program source code covered by the tests generated by each tool (50 Android apps were tested); human performance was also measured. The application source lines covered by human and tool generated tests was recorded.

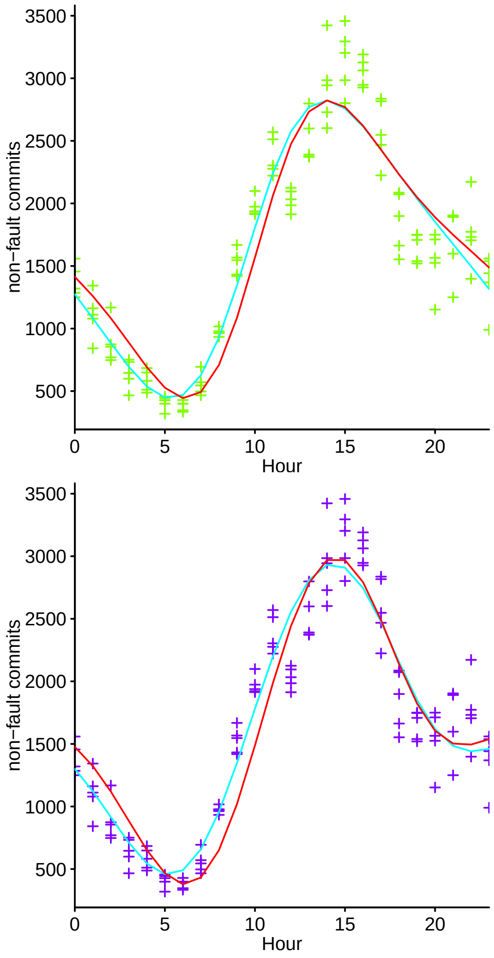


Figure 11.87: Number of commits per hour for each weekday, fitted using cos(...cos...) (upper), and cos(...cos+sin...) (lower), for Linux; in both cases the fitted fault model (red) has been rescaled to allow comparison. Data from Eyolfson et al.[559] Github–Local

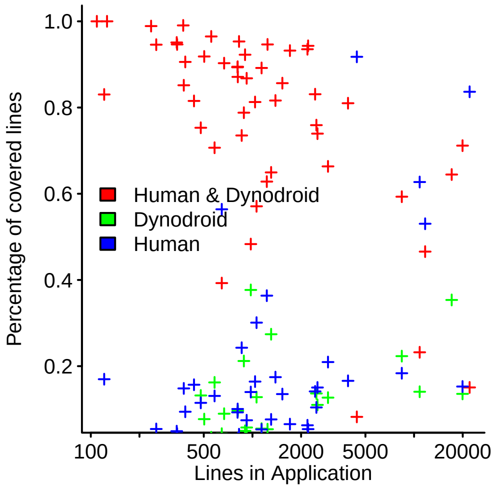


Figure 11.88: Lines of source against percentage test coverage achieved by both Human & Dynodroid tests, only by Dynodroid tests and only by Human tests, for each of the 50 applications. Data from Machiry et al.[1171] Github–Local

One measure of human vs. tool performance is to compare just those source lines that are covered by tests. What percentage, for each application, is covered by both human and tool generated tests, and what percentage uniquely covered by human or tool tests?

Figure 11.88 shows three quantities in one plot. For each of the 50 applications, the source line coverage common to human and Dynodroid tests (as a percentage of all covered lines), percentage only covered by Dynodroid generated tests and coverage of human only tests. Normalizing coverage counts, to a percentage of source lines covered, allows performance across different applications to be compared.

Fitting three regression models, one for each kind of coverage, using application source lines as the explanatory variable fails to make use of all the available information, i.e., the relationship between the three percentages. A method of combining the three percentages into a single entity, that can be used as a response variable, is required. The *isometric log-ratio transformation*, ilr, is one possibility, and the `compositions` package supports the `ilr` function.

Figure 11.89 shows the same information in a ternary plot (in blue), along with a fitted regression model (green line). The explanatory variable is application source, and the red plus signs show predictions for various totals (tick marks on the axis are measurement points where one of the three components is zero). The quality of fit is very poor, with potentially many outliers and non-constant variance; other explanatory variables, not present in the data, may enable the building of a better fitting model.

The clustering of points near the Human & Dynodroid vertex shows that tests created by these two generators tend to cover the same source lines. More points are near the Dynodroid axis than the Human axis, suggesting that Dynodroid generated tests cover fewer unique source lines.

The following code was used to fit the regression model:

```
library("compositions")

covered=acomp(dh, parts=c("LOC.covered.exclusively.by.Dyno..D.",
                          "LOC.covered.exclusively.by.Human..H.",
                          "LOC.covered.by.both.Dyno.and.Human..C."))

plot(covered, labels="", col=point_col, mp=NULL)
ternaryAxis(side=0, small=TRUE, aspanel=TRUE,
               Xlab="Dynodroid", Ylab="Human", Zlab="Human & Dynodroid")

dh$l_total_lines=log(dh$Total.App.LOC..T.)

comp_mod=lm(ilr(covered) ~ I(l_total_lines^2), data=dh) # fit model

d=ilrInv(coef(comp_mod)[-1, ], orig=covered) # extract model coefficients
straight(mean(covered), d, col="green")      # line of fitted model
```

The `acomp` function normalises the columns passed as an argument using a ratio scale, and returns an object having class `acomp` (named after Aitchison who pointed out the useful mathematical properties that a ratio scale bring to compositional analysis).

The `ilr` function is not currently handled by `glm`, so `lm` has to be used. Understanding the following code requires a lot more background knowledge than is appropriate here; see van den Boogaart and Tolosana-Deldago[1853] for more details.

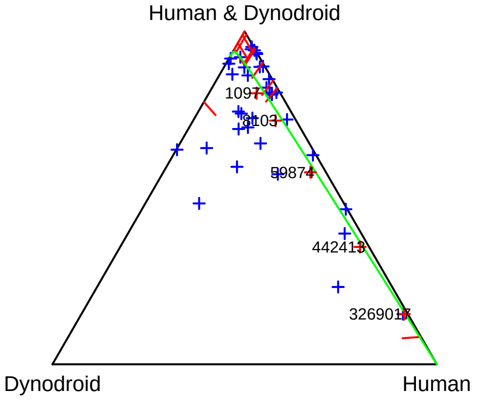


Figure 11.89: Ternary plot of composed from source lines covered by both Human & Dynodroid tests, by only by Dynodroid tests and only by Human tests (measurements in blue); fitted regression line (green) and prediction points (red) for various total source lines (numeric values). Data from Machiry et al.[1171] Github–Local

Chapter 12

Miscellaneous techniques

12.1 Introduction

This chapter covers techniques which produce results that do not have the explicit equational form available with regression models.

12.2 Machine learning

Machine learning is the name given to a collection of techniques for automatically building a black-box prediction model, learned from training examples.

Users of machine learning do not need to understand the data (although it helps if they do), and as such, this approach to model building is ideal for clueless button pushers. From time to time, we are all clueless button pushers; machine learning is an easy-to-use tool that can help find a path through the fog.

This book's emphasis is on understanding the processes involved in software engineering, not building black-box prediction models.

The quality of the predictions made by models built using machine learning, depend on the quality of the training data used. It is worth noting that: the blacker the prediction box, the faster feedback is needed on prediction accuracy. Following black box predictions, without regular feedback on their accuracy is a recipe for disaster.

A sample containing information about many variables is always useful to have; domain knowledge might be used to select an appropriate subset. However, when all the variables are included in the analysis at the same time the *curse of dimensionality* arises.

A common metric used by machine learning algorithms is the distance between points. Each measurement can be viewed as a point in an n-dimensional space, where n is the number of attributes associated with each measured item. For ease of comparison in the following analysis, every side in this n-dimensional space is assumed to have length one, and so its volume is also one. In 3-dimensions the volume of a sphere of diameter one is $\frac{4}{3}\pi 0.5^3 \rightarrow 0.52$, that is the sphere occupies 52% of the unit cube, i.e., if the unit cube contains multiple points, there is a 52% probability that a point at the center of the unit cube is within 1-unit distance of another point. As the number of dimensions increases the sphere/unit cube volume ratio increases to a peak at five dimensions, and then decreases rapidly. Figure 12.1 shows how the volume of a sphere changes, relative to the volume of the unit cube, as the number of dimensions increases.

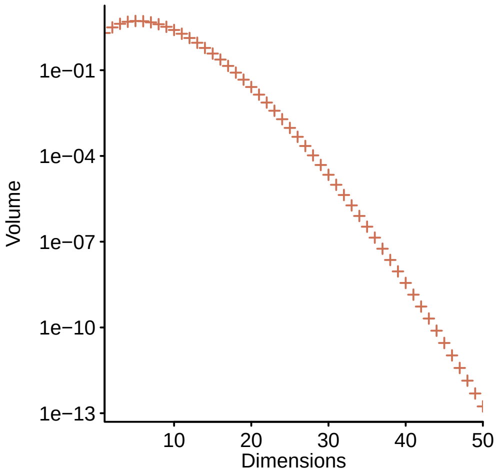


Figure 12.1: Volume of unit sphere in 1 to 50 dimensions, e.g., sphere has volume $\frac{4}{3}pi$ in three dimensions. Github–Local

As the number of dimensions increases, the distance from a point to the point nearest to it approaches the distance to the point furthest from it;[188] an effect that can occur with as few as 10-15 dimensions. This behavior means that any algorithm relying on distance between points effectively ceases to work at higher dimensions.

Text analysis Software engineering produces often produces large quantities of text written in natural language, e.g., English. Like source code, this natural language text is raw material from which useful information might be extracted.

Automated extraction of semantics from natural language is an unsolved problem, for the general case. Approximate answers can sometimes be obtained to specific kinds of semantic questions. Some algorithms are based on using prelearned examples, e.g., sentiment analysis is a popular technique for estimating whether text expresses a positive and negative opinion, but the results depend on the training data and tool used[931] (researchers are starting to collate software engineering specific training data[1134]). Dependence on training data is an important issue for any approach based on using pretrained models.

Available R packages for text analysis include tm (along with extension packages, such as tm.plugin.mail for processing emails and tm.plugin.webmining for mining web pages), and spacyr provides an interface to the spacy.io natural language processing system. For an example, see Github–faults/reopened_text.R.

12.2.1 Decision trees

As the name suggests decision tree models take the form of a tree like structure. Each node of the tree contains either an expression whose result is used to select which of two branches to follow, or a value denoting the result. The rpart[i] package supports the creation of binary decision trees.

Tree models are popular for use cases where a model is needed that can be interpreted by the people making a decision, based on what they observe, e.g., Doctors. Decision trees are the canonical example of a machine learning model that is not a black-box.

Each tree node contains a binary relationship, which selects the branch to follow to the next node, with the process continuing until a leaf node is reached. The model building process decides whether a leaf node should be split into a condition node and two leaf nodes using a method known as *cost complexity pruning*; any node split that does not improve the overall fit by a factor of CP is not attempted.

A study by Shihab, Ihara, Kamei, Ibrahim, Ohira, Adams, Hassan and Matsumoto[1674] investigated reopened faults in the Eclipse project. Of the 18,312 bug reports, 3,903 were resolved (i.e., closed at least once) and 1,530 of these could be linked to code changes. Of the 1,530 that could be linked to code changes, 246 had been reopened at the time of the study. Shihab et al cast their net very wide, extracting 22 factors that could possibly be associated with reopened faults.

Figure 12.2 shows the first few levels of a fitted decision tree, which is visibly very cluttered. The names of the people reporting and fixing problems is part of the fitted model, resulting in some overly long lines (names have been truncated to two characters, so something is visible). The rpart package provides basic plotting functionality, and rpart.plot package provides much more functionality; the following is the essential code:

```
library("rpart")
library("rpart.plot")

dt=rpart(remod ~ time+week_day+month_day+month+time_days+description_size+
             severity+priority+pri_chng+ num_fix_files+num_cc+prev_state+
             fixer_exp+fixer_name+reporter_exp+reporter_name,
        data=raw_data, weight=data_weight,
        method="class", x=TRUE, model=TRUE, parms=list(split="information"))
rpart.plot(weighted_model, cex=1.2, split.font=1, under.col=point_col,
               box.palette=c("green", "red"), branch.col="grey",
               under=TRUE, type=4, extra=100, branch=0.3, faclen=2)
```

[i] Recursive partitioning.

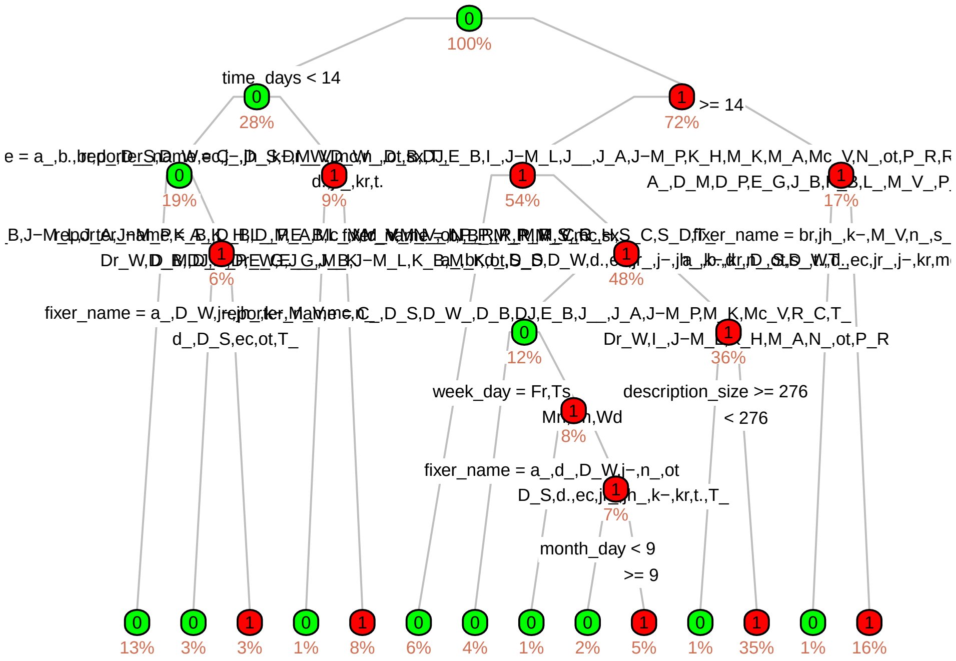


Figure 12.2: Top levels of the decision tree fitted to the reopened fault data (overly long lines are names of people who reported and fixed the fault). Data from Shihab et al.[1674] Github–Local

Which variables contribute most to the model? The summary (the cp=0.4 argument removes lots of details from the output; the printcp function does not provide any information on variable importance):

```
> summary(weighted_model, cp=0.4)
Call:
rpart(formula = remod ~ time + week_day + month_day + month +
    time_days + severity + priority + pri_chng + num_fix_files +
    num_cc + prev_state + description_size + fixer_exp + fixer_name +
    reporter_exp + reporter_name, data = raw_data, weights = data_weight,
    method = "class", model = TRUE, x = TRUE, parms = list(split = "information"))
  n= 1530

          CP nsplit rel error    xerror       xstd
1 0.17010632      0 1.0000000 1.0842714 0.01972192
2 0.02814259      1 0.8298937 0.9061914 0.01969849
3 0.02751720      2 0.8017511 0.9118199 0.01970884
4 0.02095059      5 0.6957473 0.8858662 0.01965584
5 0.01485303      6 0.6747967 0.8758599 0.01963179
6 0.01414947      7 0.6599437 0.8364603 0.01951736
7 0.01407129      9 0.6316448 0.8364603 0.01951736
8 0.01219512     10 0.6175735 0.8355222 0.01951425
9 0.01000000     13 0.5795810 0.8317699 0.01950163

Variable importance
      fixer_name    reporter_name        time_days         week_day
              32               32               13                6
description_size        fixer_exp     reporter_exp        month_day
               4                3                2                2
        priority         severity       prev_state    num_fix_files
               1                1                1                1
           month
               1

Node number 1: 1530 observations
  predicted class=0  expected loss=0.4990637  P(node) =1
    class counts:  1284 1279.2
   probabilities: 0.501 0.499
```

The variables making the largest contribution to the model, and a measure of their relative importance, appear at the end (at least when a large cp argument is passed).

For the identity of people fixing and reporting problems to play such a large role in the model, either this subset of people do work that is more likely to need to be looked at again in the future , or the faults they close have some characteristic that causes the faults they close to be reopened. This issue cannot be analyzed further using the available data.

The columns of numbers in the middle of the output contain two measures of error, for various values of CP. Decision trees are susceptible to overfitting, and the xerror column estimates the error using ten-fold cross validation (the error listed in the rel error column does not use cross validation and gives a rosier estimate). The output above suggests that building a model using the CP value listed in the second row is likely to produce more accurate results than other values (the default value of CP is 0.01).

See Github–odds-and-ends/wcre2012-delaystudy.R for an example of a decision tree analysis of data containing many variables.

12.3 Clustering

Clustering is the process of grouping together items (into one or more clusters), such that items in a given cluster are more similar to each other than items in other clusters. Some commonly used measures of similarity include distance between items, density of items, and distance from a set of items chosen to be representative of some collection of characteristics of interest.

A clustering approach to data analysis requires a method for measuring item similarity, and an algorithm for using this information to group the appropriate items within the same cluster. Examples of attempts to understand data, via clustering, include: location based distance (fig 2.19), similarity between Linux distributions based on packages they contain (fig 4.17), trading relationships between companies (fig 4.40), developers contributing to the same Apache project (fig 4.32), density of variables experiencing a given number of read/writes (fig 7.44) and correlation between attributes of Github pull requests (fig 8.9).

A study by Kanda, Ishio and Inoue[959] reverse engineered the evolutionary history of nine large software systems by comparing the similarity of the files containing the source code used to build successive releases. Figure 12.3 shows an unrooted tree representation of the phylogenetic tree estimated from the paired similarity of corresponding files contained in various releases of OpenBSD, FreeBSD and NetBSD.

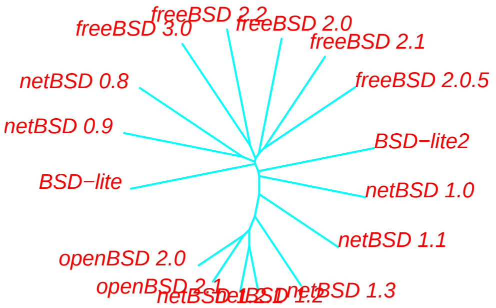


Figure 12.3: Unrooted tree denoting a phylogenetic tree estimated from the paired similarity of the corresponding source files contained in some releases of the major variants of BSD unix. Data kindly supplied by Kanda.[959] Github–Local

12.3.1 Sequence mining

Recurring subsequences may occur in a collection of related item (or event) sequences. A common application of sequence mining is recommendation systems, such as finding items that shoppers often buy together, or in sequence (e.g., as children grow up), or shared buying patterns between groups of shoppers.

Common sequences often occur in calls to a given API, e.g., open/read/close.

The arules package supports the mining of association rules and frequent item sets.

Given N sequences of items, the fraction of these sequences containing a particular item, say X, is known as the *Support* for X. Given that X appears in a sequence, what is the likelihood that Y will also appear in that sequence (written $\{X \Rightarrow Y\}$)? The following are two common answers:

$$Confidence\{X \Rightarrow Y\} = \frac{Support\{X,Y\}}{Support\{X\}}$$

$$Lift\{X \Rightarrow Y\} = \frac{Support\{X,Y\}}{Support\{X\} \times Support\{Y\}} = \frac{Confidence\{X \Rightarrow Y\}}{Support\{Y\}}$$

A study by Fowkes and Sutton[619] investigated the sequence of API calls made within the method bodies of 17 Java systems. The following call to apriori searches for sequences of method calls (in the drools business management system) having a given *support* and *confidence*; see Github–odds-and-ends/drools.R:

```
library("arules")

drools=read.transactions(paste0(ESEUR_dir, "odds-and-ends/drools.csv"),
                         format="single", cols=c(1, 2))

rules=apriori(drools, parameter=list(support=0.0001, confidence=0.1))

summary(rules)
inspect(head(rules, n=3, by = "confidence"))
```

The number of rules, of a given length, found were (lhs and rhs refer to the two sides of the ⇒ symbol):

```
rule length distribution (lhs + rhs):sizes
   2    3    4    5
1478  252   32    5
```

If many results are returned, some form of visualization can help reduce the effort needed to appreciate the distribution of results. The arulesVis package supports a variety of ways of visualizing association mining results. Figure 12.4 shows what is known as a *two-key plot* of the above results; the colored orders indicates the number of items contained in each rule.

The apriori algorithm treats each item, in a sequence, as being independent. The order of method calls may be significant, and the arulesSequences package adds sequence mining functionality to the arules package.

A required call may be missing from a sequence of method calls; the recommenderlab package provides support for developing and testing recommendation algorithms.

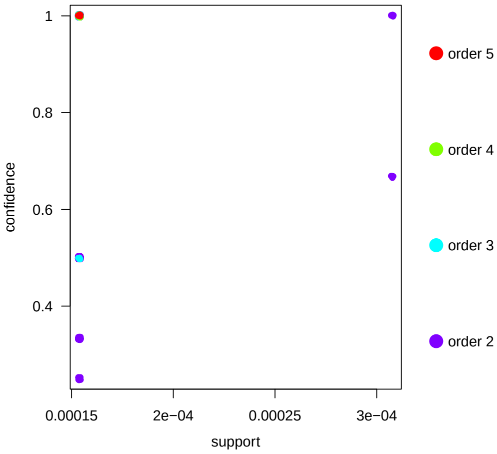


Figure 12.4: A two-key plot of associating mining results; order indicates number of items in rules. Data from Fowkes et al.[619] Github–Local

12.4 Ordering of items

Arranging items in order can reveal information about the form of the calculation used to select the relative positions of ordered items. Given multiple orderings of the same items, ordering patterns of subsequences of items, common to multiple sequences may indicate a shared semantics for those items.

A ranking is created when items are placed in an order relative to each other. Items are rated when, for instance, people are asked to provide a relative rating based on an ordinal scale, e.g., the extent to which a person like/disliked a book. The analysis of rating data is discussed in section 13.4.

12.4.1 Seriation

Seriation is the process of placing items into a linear order, based on a metric derived from some item characteristics. The number of possible item orderings grows as $n!$, making it impractical to evaluate every possibility for non-trivial sample sizes; heuristic algorithms have to be used. The seriation package supports a variety of functions that attempt to find an optimal linear ordering of items (see fig 4.32 and fig 7.10).

A study by Jones[923] investigated the extent to which developers create similar data structures to hold information listed in a specification, e.g., grouping together identifiers containing related information in the same data structure. The hypothesis was that shared cultural and professional experiences would result in subjects defining data structures containing similar contents.

Subjects were given a list of items from the “Department of Agriculture”, and asked to design a C/C++ API containing this information. The results, from each subject, were the structs or classes defined and their fields/members (with each field containing one item of API information, e.g., “Date crop harvested” and “Organically produced”).

Ordering the data highlights API information that tends to be colocated in the same data structure, and the subjects making similar choices.

Figure 12.5 shows the items placed in the same data structure as the information item “Antibiotics used”, by each subject (colored squares indicate presence). The matrix passed to seriate contains boolean values (indicating presence in the same data structure), and the subjects/fields are ordered so that shared usage appears adjacent. The bertinplot function provides a particular visualization of the ordered data.

```
library("seriation")

fser=seriate(fmat, method="BEA",  control = list(rep = 10))
bertinplot(fmat, fser, options=list(panel=panel.squares, spacing=0,
                                    gp_labels=gpar(cex=0.6)))
```

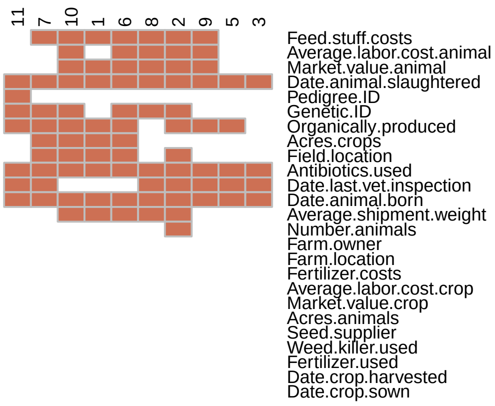


Figure 12.5: A Bertin plot for items included in the same data structure as the item “Antibiotics used”, for each numbered subject, after reordering by seriate. Data from Jones.[923] Github–Local

A single item's pattern of association, with all the other items, can be generalised by counting occurrences of every pair of items in the same data structure. A Robinson matrix has the property that the value of its matrix elements decrease, or stay the same, when moving away from the major diagonal; this matrix has been used to study commonality in subjects' categorization behavior.[1583] Figure 12.6 shows a visualization of a Robinson matrix for the Jones data.

```
library("seriation")

fdist = as.dist(1 - fmat/max(fmat)) # Normalise counts
fser = seriate(fdist, method="BBURCG")

pimage(fdist, fser, col=pal_col, key=FALSE, gp=gpar(cex=0.8))
```

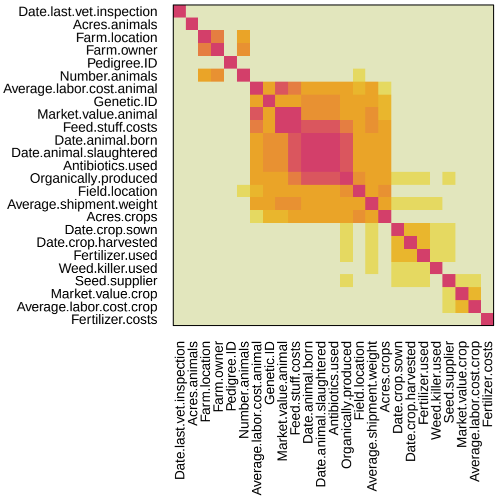


Figure 12.6: A visualization of the Robinson matrix based on number of times pairs of items co-occur in the same data structure (the closer to the diagonal the more often they occur together). Data from Jones.[923] Github–Local

12.4.2 Preferred item ordering

A list of items may have a preferred ordering. The ordering may be the result of ranking, rating or a comparison between pairs of items.

Bradley-Terry statistics are a traditional technique for calculating an ordering for a list of items, based on results from pairwise comparisons, e.g., football match results (there is an extension for analyzing results from simultaneous comparisons of more than two items). This section is based around use of the BradleyTerry2 package for simple paired comparisons, and the PlackettLuce package for the more complicated cases.

An alternative approach to analyzing item ordering is discussed in section 9.6.1.

Given a *contest* between i and j, the probability that i beats j is assumed to have the form:

$P(i > j) = \frac{\alpha_i}{\alpha_i + \alpha_j}$, where α_i and α_j might be thought of as some measure of *ability* of i and j.

Bradley-Terry statistics uses logistic regression to model this equation, and fits the βs in the following equation:

$$P(i > j) = \frac{e^{\beta_i}}{e^{\beta_i} + e^{\beta_j}}, \text{ where: } \alpha_i = e^{\beta_i} \text{ and } \alpha_j = e^{\beta_j}.$$

A study by Jones[920] investigated developer beliefs about binary operator precedence. Subjects saw an expression, such as `a + b % c`, and were asked to insert parenthesis such that the behavior (as interpreted by the compiler) remained unchanged. Based on subject answers, what is the relative precedence of the binary operators used in the study (which may be different from the actual precedence)?

The parenthesis specifies the operator that can be treated as *winning* a precedence contest. The BTm function, in the BradleyTerry2 package[1834] takes the results from paired comparisons, and returns the coefficients of a fitted model (internally, the glm.fit function is used to fit a logit regression model).

```
library("BradleyTerry2")

prec_BT=BTm(cbind(first_wl, second_wl), first_op, second_op, data=nodraws)

summary(prec_BT)

# A less interesting plot than the one specifically created
plot(qvcalc(BTabilities(prec_BT)), col=point_col, main="",
        xlab="Operator", ylab="Relative order")
```

The output produced by summary has the same form as that produced for other regression models. Note: the summary function does not list the factor with value zero (the subtraction operator, for this data). The BTabilities function lists all factors and their values, and the call to plot uses this information to visualize the estimated coefficients and their corresponding standard error.

Figure 12.7 shows the estimated β coefficients (along with a corresponding standard error), and can be used to estimate subjects beliefs about relative binary operator precedence. The probability of, for instance, equality, ==, *winning* a precedence choice against binary plus, +, is (based on the values returned by the fitted model): $\frac{e^{-2.08}}{e^{-2.08}+e^{0.26}} \rightarrow 0.088$, and the probability of binary plus, +, *winning* against == is: 0.912.

In a sports match, the home team is often considered to have an advantage over the away team. Perhaps, when developers are uncertain they are more likely to select the first binary operator, of a pair. A model can include information on first position *wins*, for each operator pair (the effect is very small for this data); see Github–developers/jones_prec.R for details.

The two item model can be extended to where three or more items are ranked by ordering them according to some preference criteria. For instance, for a ranking of three items:

$$P(i > j > k) = \frac{\alpha_i}{\alpha_i + \alpha_j + \alpha_k} \times \frac{\alpha_j}{\alpha_j + \alpha_k}$$

The PlackettLuce package supports the analysis of rankings of more than two items, tied ranks (i.e., items having the same rank), and rankings where some items do not appear in every ranking.

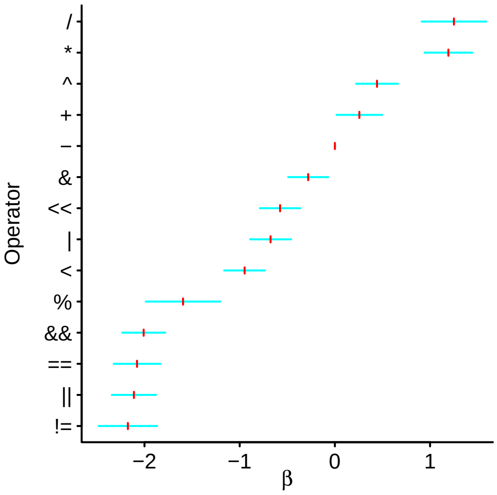


Figure 12.7: Relative ordering of binary operator precedence (i.e., value of β), and corresponding standard error, based on subject responses to binary operator precedence questions. Data from Jones.[920] Github–Local

A study by Biegel, Beck, Hornig and Diehl[193] investigated the ordering of definitions appearing in 5,372 classs, from 16 Java programs. The Java coding conventions (JCC)[1774] recommends a particular ordering of the four kinds of definitions, and on average over 80% of classes in these projects follow the JCC ordering recommendation: *class variable* (or *field*), *instance variable* (or *static initializer*), *constructor* and *method*.

Java definitions include access control information, via the use of the keywords: private, protected, public and no keyword (the default behavior given in the language specification is used). The ordering of definitions in the source code, by identifier visibility (i.e., access control), can be treated as a ranking. The declarations in a source file may not contain an instance of every kind of access control, i.e., some rankings will include a subset of items.

Figure 12.8 shows the relative ordering (ranking) of method definitions by access control keyword, over all projects investigated; see Github–sourcecode/member-order/vis-key_order-pref.R, which also includes details of other kinds of declarations. If the methods defined in a class have three kinds of visibility (only 3% of cases in the study), the probability of, for instance, the visibility order being public, protected, then private is (using β values from the fitted model):

$$\frac{e^{0.36}}{e^{0.36}+e^{-0.36}+e^{-0.67}} \times \frac{e^{-0.36}}{e^{-0.36}+e^{-0.67}} \rightarrow 0.54 \times 0.58 \rightarrow 0.31$$

When teams are ranked, with varying team membership, the hyper2 package may be of use in obtaining a ranking of the individuals.

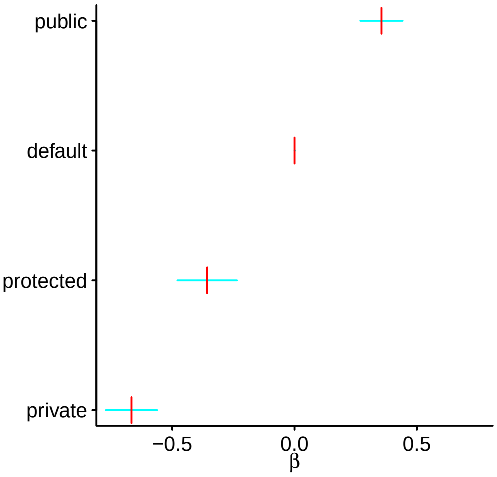


Figure 12.8: Fitted values of β for access control (visibility) of method definitions within a Java class. Data from Biegel et al.[193] Github–Local

12.4.3 Agreement between raters

A measurement may be based on human judgement, e.g., assigning a product rating. Different people may make different judgements of the same characteristic/entity, and a way of evaluating the agreement between the different judgements is needed.

Cohen's Kappa is a measure of inter-rater agreement between two raters, it varies from zero (no agreement) to one (perfect agreement). Fleiss's Kappa is a measure of inter-rater agreement between three or more raters.

The kappa2 function, in the irr package, supports Cohen's Kappa (weighting is supported by passing the argument weight="squared"); the kappam.fleiss function calculates Fleiss's Kappa.

A study by Schach, Jin, Yu, Heller and Offutt[1618] categorised the kinds of maintenance activity performed on various systems. Table 12.1 lists the categories assigned by two raters to 215 maintenance categories involving the first 20 versions of Linux. The Cohen's Kappa for these two raters is 0.805; see Github–group-compare/agreement.R.

12.5 Simulation

Simulating a process or system, via an executable model, is a means of gaining information about the operational characteristics of the process or system (assuming the characteristics of the model are sufficiently accurate). Varying the characteristics of a simulation

	Adaptive	Corrective	Perfective	Other	Total
Adaptive	2	0	0	0	2
Corrective	0	82	16	0	98
Perfective	0	5	99	2	106
Other	0	0	0	9	9
Total	2	87	115	11	215

Table 12.1: Maintenance categories assigned by two raters (row and column) for the first 20 versions of the Linux kernel at the change-log level. Data from Schach et al.[1618]

model can provide possible answers to what-if questions. Some of the kinds of simulation methods available include:

- discrete event simulation: the system modeling is based on the discrete events that can occur; supported by the `simmer` package, e.g., staff scheduling, see Github–projects/impl-sim.R,
- agent-based modeling: a set of agents, having specified attributes, interact with each other in a computer simulation. After a given number of time steps the state of the system is measured, with the results from many simulation runs combined to calculate probabilities for the various end-states. NetLogo is a widely used system for agent-based modeling and the `RNetLogo` package provides an `R` interface,
- system dynamics: represents a system using causal loops between all the interacting components, essentially an analogy representation of differential equations. This approach has been used to simulate software project staffing,[3,269,1175]
- differential equations: if a system can be described using a set of differential equations, the `deSolve` package can be used to numerically solve these equations.

  Fitting sales data to the Bass diffusion model was discussed in section 3.6.3. Duggan[506] investigated the impact, on sales volume, of the spatial separation of potential customers living in 10 regions (figure 12.9 shows the connections between regions, and the percentage of total population they contain, given as initial conditions in the numerical solution); see Github–odds-and-ends/RJ-2017.R for implementation details.

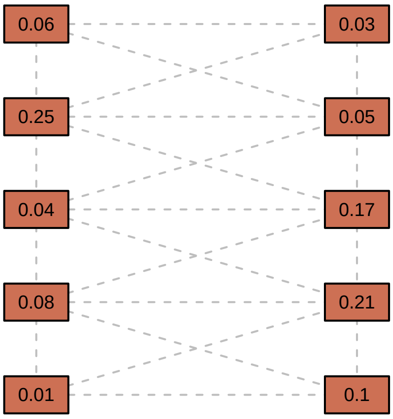


Figure 12.9: Region populations and their connections: initial conditions used in Duggan's[506] numerical solution of the Bass equation. Github–Local

Chapter 13

Experiments

13.1 Introduction

Does doing X have a significant effect on S? Traditionally X might have been a new kind of fertiliser (or drug) and S the crop yield (or being cured of some illness). In software engineering the effect sought is often a performance improvement, and X the latest snake oil. Detecting discontinuities in data is discussed in section 11.2.9.

In an observational study the researcher is a passive observer, simply recording what happened, or is happening. In an experimental study the researcher actively attempts to control the values of the explanatory variables (a common technique is to vary the values of one explanatory variable, while the others are held constant; in some cases, the environment in which events occur form a natural experiment, in that the explanatory variables of interest vary in a way that an experimenter might vary them).

A controlled experiment is the technique used to obtain the data needed to test a hypothesis. The controlled experiment that most developers are likely to be familiar with is benchmarking.

Dramatic changes in performance, after doing X, are relatively common in software development, and in such cases using statistics to confirm that a noticeable change has occurred is almost a formality. At the other extreme, differences that require statistical analysis to be detected are often not worth being concerned about in practice.

Advice for running experiments often follows the waterfall model of software development, with lots of upfront planning and little or no feedback from actual use until the end of the process. This advice has its roots in the environment in which experiments are carried out by the readership of many statistics text books, where running an experiment is costly (in money or time), or is a once only opportunity.[i]

Some experimental questions in software engineering are amenable to iteration. Running quick, inexpensive experiments, can be a cost effective technique for filtering possible questions of interest, and obtaining information on which, of the myriad of variables, have a worthwhile impact on the response variable(s).

Finding the right question to ask is sometimes the most useful output from running an experiment.

An important point to remember is, that, it is better to have an inexact answer to the right question, than an exact answer to the wrong question.

Like all software development activities, experiments have to pay their way. Some of the answers needed for a cost-benefit analysis include: the cost of running an experiment, capable of producing the information of interest, within acceptable confidence intervals; the usefulness of the data likely to be obtained, by running an experiment, using a given amount of resources (such as time and money).

Many software engineering tasks are performed within complex environments. Controlling and measuring all the variables in the environment has been perceived as being time-consuming and expensive that few researchers have been willing to attempt realistic controlled experiments. Consequently, much of the hypothesis testing performed in commer-

[i] Experiments in the social sciences, major producer of experimental studies, are often grant funded, with limited opportunities for rerunning experiments that failed to produce data that can be published.

cial environments has been based on convenience samples, obtained from experimentally uncontrolled, production software projects.

Running an experiment with minimal funding means that experimental subjects are often unpaid volunteers, from a pool containing who ever is available.

Software engineering is not known as a research area where experiments are commonly performed. A study[1696] of 5,453 papers in software engineering journals, published between 1993 and 2002, found that only 1.9% reported controlled experiments (of which 72.6% used students, only, as subjects), and the statistical power of many of these experiments fell below expected norms.[515]

13.1.1 Measurement uncertainty

The term *measurement error* is often applied to measurements involving physical quantities. It is based on the assumption that any different between the measured and actual value is caused by errors made by the measurement process.

In software engineering, some quantities can be measured exactly, e.g., lines of code in a source code file. However, the process that generated the code (e.g., a software developer) may produce a different number of lines, if repeated using a different developer or even the original developer (reimplementing the program); see fig 5.23. The code that is measured is a sample drawn from a population, and measurements involving this code needs to be treated as having an accompanying uncertainty.

Goodhart's law is an observation about human behavior, rather than a law: "Any observed statistical regularity will tend to collapse once pressure is placed on it for control purposes." If the measurements collected were actively used to control or evaluate the development team (for instance), then developers have a motivation to cause the measurements to move in a direction favorable to themselves.

A study by Perry, Staudenmayer and Votta[1452] investigated various software development activities (e.g., working time on an activity, and activities performed throughout the day), as measured by the developers involved (i.e., self reports) and as measured by external observers. Differences in reported measurement values included, developers reporting activity time 2.8% higher, on average, than that reported by an observer; the measurement agreement rate for activities performed varied between 0.6 and 0.95.

A series of studies[984] of social network data, as reported by those within the network and extracted from externally observed information, found that differences were large enough to render invalid any analysis of a network characteristics based on member supplied information.

Random variability in the performance of, what are intended to be, identical hardware components, is discussed in section 13.3.2.1.

Different tools, or different tool options may produce different results, e.g., the diff algorithms supported by Git,[1375] reported statement coverage,[1970] or clone detection technique.[1814]

A study by Li[1118] investigated the call graphs built by four Python tools. Figure 13.1 shows the number of nodes in the call graphs built by four tools, broken down by number of nodes common to each tool.

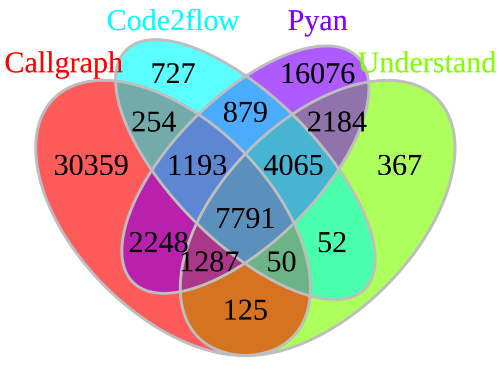


Figure 13.1: Number of nodes in the Python call graphs built by four tools, broken down by number of nodes common to each tool. Data from Li.[1118] Github–Local

Programs sometimes consume increasing amounts of memory, the longer they are run; sometimes known as *software aging*. One consequence of changes in the behavior of the environment in which a benchmark is executed, is to introduce systematic noise into the results.

A study by Cotroneo, Iannillo, Natella and Pietrantuono[403] investigated the impact of running two applications on three versions of Android, installed on four different phones, running with low/high available memory, and three kinds of event interactions (i.e., $2*3*4*2*3$ combinations of App, environment, and usage); there were an average of 272 App executions per combination, i.e., 39,187 total executions. A fitted regression model finds that the amount of memory used consistently changed by a small amount with each successive App execution, with the amount depending on environment and event use (less than 10 bytes); see Github–benchmark/2005-11523.R.

13.2 Design of experiments

Randomization is the foundation on which any claims of causation, involving experimental results, are based (i.e., doing X caused Y). Without randomization, the most that can be said is that a correlation has been observed between doing X, and Y occurring.

The purpose of running an experiment is to obtain data that can be used to help answer one of more questions. Experiments have to be carefully designed, to ensure that the data obtained is representative of the processes involved for the question(s) being asked. Design issues include:

- recruiting the appropriate subjects from the available population (sampling is discussed in section 10.2),
- creating an experimental task(s) that shares all the important characteristics of the tasks associated with the questions of interest,
- creating an environment for the subjects, in which they can give a suitable performance, during the experiment.

  Subject characteristics can sometimes interfere with good experimental design, e.g., human subjects have memories of their previous experiences that they cannot choose to erase,
- controlling all variables that could have a significant impact on the response variable(s) of interest. Failure to take into account, and control, variables having a significant impact, can cause a tiny effect to appear to be a large effect and vice versa. One person's tongue-in cheek-advice on how to bias an experiment to get the desired outcome[1271] is another person's list of thoughtless mistakes.

  When factors cannot be controlled, they need to have their impact contained. One technique for handling the problem of uncontrolled variables is to group subjects into blocks based on the variable that is suspected of influencing the response (a process known as *blocking*), randomization of subjects then occurs within each block. The identity of the block becomes another explanatory variable during analysis of the results.

A study by Basili, Green, Laitenberger, Lanubile, Shull, Sørumgård and Zelkowitz[136] compared the performance of subjects when using perspective based reading (which instructs reviewers to read a document from a specified perspective, e.g., a designer, tester or user), against the reading technique currently used by the professional developers who were the subjects.

The researchers thought it likely that, training subjects to use the new technique would alter their performance when using whatever technique they currently used, and decided to measure subject performance using their existing review technique first, before giving subjects training in the new, perspective based reading, technique.

Having all subjects use the perspective-based reading technique second, means it is not possible to separate out ordering effects in the results, e.g., effects such learning during the experiment, and any random distraction effects that only occurred at certain times.

Factors outside the control of these researchers, which could affect the results, include:

- the time taken for subjects to become proficient at using a new technique; old habits die hard. How much practice do subjects need, for them to be able to give a performance that makes it possible to reliable compare the new technique? In this study subjects were taught PBR two days after the first part of the experiment, they trained on a test document, reviewed one document, received more training and then reviewed another document.
- the kind of review technique used by subjects in the first half of the experiment. A change in performance is expected, but it is not known what technique any change is relative to (it is assumed that adhoc techniques are being used),
- the characteristics of the seeded faults. Were more faults found in the NASA documents because readers were familiar with reading that kind of document, or perhaps the characteristics of the seeded faults was such that they were harder to detect in one kind of document than another?

The experimental output included, for each subject, the number of faults detected (which has a known upper limit, and a yes/no detection status), and the number of false positives in each document reviewed (which has no upper limit, in theory); see Github–faults/basili/pbr-experiment.R for details of fitting a regression model.

Basing all experimental choices on random selection does not automatically create samples that maximise the information that can be obtained.

A study by Porter, Siy, Mockus and Votta[1487] investigated software inspections. The structure of the inspection process was manipulated by varying the number of reviewers (1, 2 or 4), number of meeting (1 or 2), and for multiple meetings whether reported faults were repaired between meetings (88 inspections occurred, involving 130 meetings and 17 reviewers).

Selecting the treatment to use, for a review, from successive entries on a randomised list of all possible treatment structure combinations (created at the start of the study), would ensure that the results are balanced across the variables of interest. However, in this study the choice of treatment to use was randomly selected from all possibilities, as each unit of code became available for review, resulting in some combinations of reviewers/meetings/repaired not being used, and some used very often. The results contain an unbalanced set of experimental conditions, making it difficult to fit a reliable model; see Github–experiment/porter-siy/inspection.R, Github–experiment/porter-siy/meeting.R and fig 11.33.

The complexity of computing platforms means their behavior can quickly change. A study by Barrett, Bolz-Tereick, Killick, Mount and Tratt[132] investigated the performance of Just-in-time compilers. The computer system (three different systems were measured) was rebooted and a benchmark run 2,000 times, this process was repeated 30 times (i.e., 60,000 executions of every benchmark on three systems). Every effort was made to reduce measurement noise, e.g., as many background processes as possible were disabled.

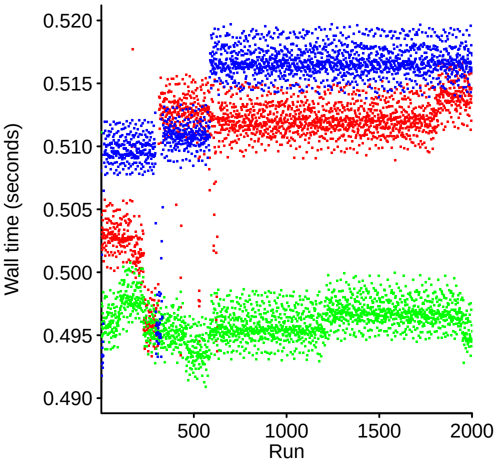


Figure 13.2: Time taken by 2,000 runs of a Javascript BinaryTree benchmark, with JIT enabled, on a quad-core Intel i7-4790; three colors are three iterations of the process: reboot machine, execute 2,000 runs. Data from Barrett et al.[132] Github–Local

Figure 13.2 shows the wall time taken for three sequences of 2,000 executions of a Javascript BinaryTree benchmark, running on a quad-core Intel i7-4790. The abrupt changes in performance match a change in the processor core used to execute the code; the benchmark process could have been locked to a given core, but would that be representative of real-life use? As discussed later (see fig 13.21) performance variability between system reboots can be larger than between a sequence of runs during one uptime.

13.2.1 Subjects

Experimental subjects might be people or artefacts (e.g., hardware or source code). An essential requirement for generalising the results from an experiment, to a larger population of subjects, is that the characteristics of the sample of experimental subjects are representative of the applicable characteristics of the population of interest. Statistical issues around sampling are discussed in section 10.2.

The major issues involved in having computer hardware as an experimental subject are covered in section 13.3, while the major issues in human cognitive performance are covered in chapter 2. A limiting factor, when designing an experiment involving human subjects, is typically the amount of time the subjects are likely to be willing to make available to participate.[1695]

When people are the subjects in experiments, a variety of human factors introduce uncertainty into the results; people constantly adapt to their environment, including the environment of an experiment (e.g., they learn and retain memories of their experiences; see section 2.5), they also experience fatigue, and their attention ebbs and flows during an experiment.

Professional developers, working within different ecosystems, may share a set of basic skills and knowledge, such as being able to fluently use at least one programming language.

Much of the published research involving human subjects, in software engineering experiments, has used students. Students are a convenience sample for many researchers, with results based on student subjects being accepted for publication by some journals. Industry is well aware that students' software engineering skills are not representative of professional developers (who have a few years experience); industry is where many graduates find employment after graduation, and the abilities of these new employees is plain for everyone in industry to see.

- students' commercial software skills and knowledge is likely to be very poor, in comparison to professional developers.[1308] This lack of experience and know-how means that student subjects need to spend time on activities that are second nature to professionals, or they simply make noncommercial judgement calls,

- students, typically, have very little experience of writing software, perhaps 50 to 150 hours (and many have no basic coding skills[1140, 1218, 1847]), while commercial software developers are likely to have between 1,000 to 10,000 hours of experience. This lack of programming fluency means that student programming performance is likely to contains a large learning component, as well as student performance being much lower than professional developers,[1288]

In other research areas students subjects may be more representative of the target population, because they have had many years of experience performing the activities and tasks used in those areas, e.g., processing text written in English and everyday image processing, which are activities used in cognitive psychology experiments.

When experimental results are intended to be applied to the population of university students studying a software related subject, subjects drawn from this population can be representative.

In the US, UK and some other countries, students pay to attend university, and like all businesses universities have to respond to customer demand. When a student decides to study a computing related subject, the University's interests are in ensuring that the student meets its minimum entry requirements and can pay; the likelihood of that person being offered employment in a software related job is not a consideration (in the UK computer science graduates have a much higher unemployment rate, six-months after graduation, compared to those studying other STEM subjects[1652]). Given the high failure rate for computing degrees[1958] and introductory programming courses,[1913] many students on such courses may not even have any software development skill or ability.

Note on terminology: many academic studies use the phrase *expert* to describe subjects who are final-year undergraduates or graduate students, with the term *novice* used to describe first-year undergraduates. In a commercial software development environment a recent graduate is considered to be a *novice* developer, while somebody with five or more years of commercial development experience might know enough to be called an *expert*.

Amazon's Mechanical Turk is becoming popular as a resource for finding subjects and running experiments (in 2015 the population of workers was estimated to be 7,300[1759]). Subjects can stop taking part in a MTurk experiment at any time and care needs to be taken to ensure that the characteristics of subjects who remain does not bias the results.[2000]

Instances of source code can be measured exactly, but different people write different code, i.e., measurements of source code contain implicit variability; see figure 5.23.

13.2.2 The task

Generalizing experimental results to daily work conditions, requires that the characteristics of the tasks performed by subjects share the essential characteristics of the work tasks. That is, the task needs to mimic realistic activities (the technical term is being *ecologically valid*):

- being representative of real world intended usage requires information about how a system will be used in practice, along with the inputs it is likely to experience; lack of resources to perform an analysis of real world usage often means that a convenience sample is used. In a rapidly changing environment, it may not be possible to specify usage patterns in sufficient detail, and perhaps one of the important real world behaviors that needs to be benchmarked is adaptability to change.

  A study by Gregg and Hazelwood[730] provides an example where data usage characteristics are the deciding factor in a cost/benefit trade-off. The time taken to perform a matrix multiply, when the CPU uses a local GPU (the SGEMM implementation in the nVidia CUBLAS package[1378] was used) was measured. Figure 13.3 shows that moving data between the CPU and GPU consumes a significant amount of time, relative to the work done on the data once inside the GPU. Estimating whether GPU usage is worthwhile, depends on the size of matrices encountered in the real world use case, the performance may be slower because of the data transfer overhead, or faster if the matrices are large enough to consume the larger amount of compute resources.

  Another example of the impact of variations in the input data relates to fig 10.9,

- Products that appear to be very similar can have very different performance characteristics (this is one reason why they exist as different products; the other common reason is marketing). Using a product that is identical to the one used in production avoids this problem.

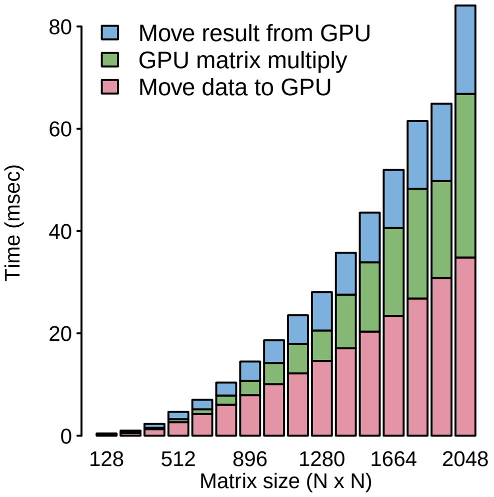
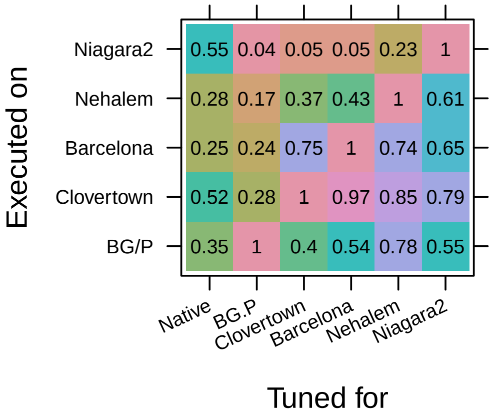


Figure 13.3: Time taken to transfer and multiply 2-dimensional matrices of various sizes on a GTX 480 GPU. Data kindly supplied by Gregg.[730] Github–Local

In a study by Bird,[198] a performance optimization expert took the existing generic code of a library and created tuned versions for each of five different processors (IBM's Blue Gene P and four different members of Intel's x86 product line). The performance of the generic, and all tuned versions of the code, was measured on all processors. Figure 13.4 shows relative performance, with the x-axis listing the processor the code was tuned for, and the y-axis the processor on which it was run; the numbers are relative performance difference, compared to running code on the processor for which it was specifically written.



Executed on	Native	BG.P	Clovertown	Barcelona	Nehalem	Niagara2
Niagara2	0.55	0.04	0.05	0.05	0.23	1
Nehalem	0.28	0.17	0.37	0.43	1	0.61
Barcelona	0.25	0.24	0.75	1	0.74	0.65
Clovertown	0.52	0.28	1	0.97	0.85	0.79
BG/P	0.35	1	0.4	0.54	0.78	0.55

Figure 13.4: Relative performance (y-axis) of libraries optimized to run on various processors (x-axis). Data from Bird.[198] Github–Local

Availability of resources is often a constraining factor, for running an experiment that mimics real world usage. For example, benchmarking backup/restore tools, or desktop search applications, requires realistic file system contents (e.g., the file system must contain a realistic number of files, directory depth, disk fragmentation, etc); getting to a position of being able to generate realistic file systems is a non-trivial task,[15] let alone realistic file content characteristics.[1794]

13.2.3 What is actually being measured?

Subjects may not solve the problems they are presented with, in an experimental, in ways that were intended by the person who designed the experiment.

The history of research into human memory provides an example of how early experimental results were misinterpreted.[917] These early experiments asked subjects to remember sequence of digits, the results suggested that short term memory has a capacity limit of 7 ± 2 items.[1267] After many years and more experiments, the 7 ± 2 digit limit model was replaced by a model based on a limit of 2 seconds of sound[109] (in English this corresponds to around 7 digits, 5.8 in Welsh[534] and around 10 digits in Chinese:[841] the number of digits that can be held in memory when people use these languages).

Software developers are problem solvers and get plenty of practice in finding patterns that can be used to achieve a goal. Unless an experiment is carefully constructed, it is naive to assume that developers will use any of the techniques anticipated by the person designing the experiment.

Your author once ran several experiments,[920] expecting to find a *two seconds of sound* effect in developers short term memory of source code (sequences of simple assignment statements were used). A great deal of effort was invested in creating code sequences whose spoken form required either more or less than two seconds of sound, but the results did not contain any evidence of the expected effect (i.e., a difference in performance caused by the length of sound in the spoken form of source code statements).

At the end of one experiment, a subject mentioned a strategy used to help improve his performance: remembering the first letter of each variable (your author had not noticed that the variables in each list had unique first letters). Use of this strategy reduced the amount of STM the subject needed to use, providing one explanation why the expected effect was not found. The last task, on subsequent experiments, asked subjects to list any strategies they had used during the experiment.

What appear to be small differences, can have a large impact. Human written source code contains very similar constructs where, what might be thought to be small differences in semantics, have usage patterns which are very different. For instance, many languages allow numeric literals to be specified using decimal and hexadecimal notation; figure 13.5 shows that the distribution of literal values written using each notation is different (at least in C source).

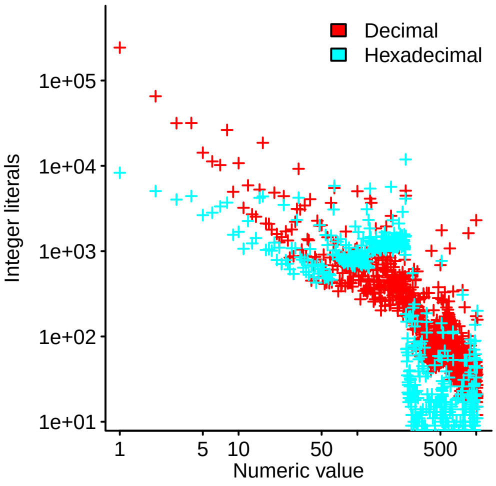


Figure 13.5: Number of integer constants, appearing in the visible form of C source code, having the lexical form of a decimal-constant (the literal 0 is also included in this set) and hexadecimal-constant that have a given value. Data from Jones.[919] Github–Local

A study[745] of how subjects split identifiers, in source code, into components, and expanded them, or not, into words, measured individual performance against what was considered to be the definitive expansion of each identifier. The results of this experiment could also be used to measure the performance of the researchers, in creating a list of identifier expansions that maximised the likelihood of developers correctly decoding the intended information.

Subject motivation is an important factor in obtaining reliable experimental data. Subjects who feel they are being coerced may respond by providing spurious responses, or simply attempt to find short-cuts that minimise the time then need to spend taking part in the experiment, without attracting attention by making too many mistakes.[784] Your author always requests that subjects put as much effort into performing the task, as they would at work: not more, not less.

13.2.4 Adapting an ongoing experiment

The costs of running an experiment makes it tempting to stop, as soon as what is thought to be enough data has been obtained (to produce a sufficiently reliable result). For instance, a researcher may process subjects in batches, running statistical tests after each batch of results, to find out whether the numbers look good/bad enough to stop. One study[181] of A/B testing estimated that 73% of experimenters stopped their experiment once a 90% confidence level was reached.

As each subject is analysed, differences in subject performance cause the aggregated values to fluctuate, and it is possible that some cut-off value (e.g., a p-value cutoff level) is achieved; however, later data may causes it to fluctuate to a less extreme value.

When running an experiment on a live system (or on a stream of subjects), an analysis of the ongoing measurements may suggest that certain changes to the experiment may have a worthwhile impact. Adaptive designs is the term used for experimental designs that support modification of an experiment as results become available.

An adaptive design might be used to reduce the number of different combinations that need to be measured, e.g., benchmarking a computing system supporting many options.[1514]

13.2.5 Selecting experimental options

The behavior of some systems may be configured by selecting from a variety of options; an estimate of the impact of individual options, on system performance, may be required. One way of obtaining this information, is to measure system performance for all possible combinations of option values. This approach might be practical for a few options, each having relatively few values, e.g., Apache supports nine build time yes/no options giving 2^9 possible configurations (out of these 512, only 192 are valid). However, large systems often support so many options, that building and executing every configuration would be impractical (e.g., SQLite supports 3,932,160 valid options).

An experiment in which all possible permutations of option values are tested, is known as a *full factor design* (options go by the experimental term *factors*). The `fac.design` function, in the `DoE.base` package, takes a specification of factor levels and returns a list of all combinations that need to be run to perform a full factor design; see Github–experiment/design_fac.R.

A study by Citron and Feitelson[358] investigated the performance impact of adding, what they called a Memo-Table (essentially a cache designed to store and reuse the results of previously executed instruction sequences), to the IBM Power4 cpu architecture. The configuration options for the Memo-Table were: Size (1k or 32k), Associativity (1-way or 8-way), Mapping (indexing by program counter or operand+opcode) and Replacement method (random or least recently used).

Four configuration parameters, each having two possible values, gives $4^2 \rightarrow 16$ possible configurations. Citron and Feitelson benchmarked all 16 possibilities, enabling them to check for interactions between all factors. In many experiments the number of interactions between factors is small, and a common cost saving is to only consider interactions between pairs of factors (rather than, say, between three factors).

Factor having just two possible values is a common case, and is known as a two-factor factorial design: it is a *full two-factor design* when all combinations used, and a *fractional two-factor design* when a subset is used.

A study by Lee and Brooks[1091] involved three optional values per parameter; see Github–experiment/lee2006/lee.R.

The `FrF2` function, in the `FrF2` package, generates a list of the combinations of factor values that need to be run, to analyse N factors having a resolution of R (the ability to separate out main effects and interactions between factors; to be able to separate out main effects a resolution of 3 is required, a resolution of 4 enables detection of separate pairs of interactions). In the following list, `1` indicates the option is enabled, `-1` that it is disabled; the output from some functions uses +/-, rather than `1`/`-1`:

```
> library("FrF2")
> FrF2(nfactors=4, resolution=3, alias.info=3)
   A  B  C  D
1  1 -1  1 -1
2  1 -1 -1  1
```

```
3  1  1 -1 -1
4 -1  1  1 -1
5 -1  1 -1  1
6 -1 -1  1  1
7  1  1  1  1
8 -1 -1 -1 -1
class=design, type= FrF2
```

The price paid for running a fractional, rather than full, factorial design experiment, is that it is not possible to distinguish interactions between some combinations of factors. For instance, after running the eight combinations listed above, it is not possible to distinguish between an effect caused by a combination of the `AB` factors, and one caused by the combination `CD`; this combination is said to be *aliased*. The complete list of aliased factors is:

```
> design.info(FrF2(nfactors=4, resolution=3, alias.info=3))$aliased
$legend
[1] "A=A" "B=B" "C=C" "D=D"

$main
[1] "A=BCD" "B=ACD" "C=ABD" "D=ABC"

$fi2
[1] "AB=CD" "AC=BD" "AD=BC"

$fi3
character(0)
```

To distinguish between an effect caused by any of these combinations, all 16 factor combinations have to be run.

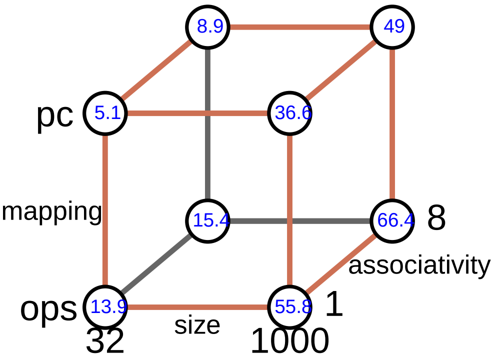


Figure 13.6: A cube plot of three configuration factors and corresponding benchmark results (blue) from Memory table experiment. Data from Citron et al.[358] Github–Local

Factorial designs require the number of runs to be a power of two, so the number of different runs grows very quickly as the number of factors increases.

A Plackett and Burman design requires that the number of runs be a multiple of four and at least one greater than the number of factors (they are non-regular fractional factorial 2-level designs). The down-side of these designs is that the results from experiments using them will only support the analysis of the main factors, i.e., any interactions between factors will not be detected; also Plackett and Burman designs can contain complex aliasing between the main factors and (possible) interactions between pairs of factors. The `pb` function, in the `FrF2` package, generates Plackett and Burman designs.

Multiple fractional factorial designs can be combined to isolate effects (i.e., remove aliasing between combinations of factors). Some signs in the original design are switched, creating what is known as a *fold over* of the original; switching the signs of all factors is known as a *full fold over*. The `fold.design` function generates a foldover design from an existing design.

Randomly selecting option values is an inefficient use of resources, because some option values are over/under used (see Github–experiment/SQL$_P$*WR.R* from a study by Guo et al[751]).

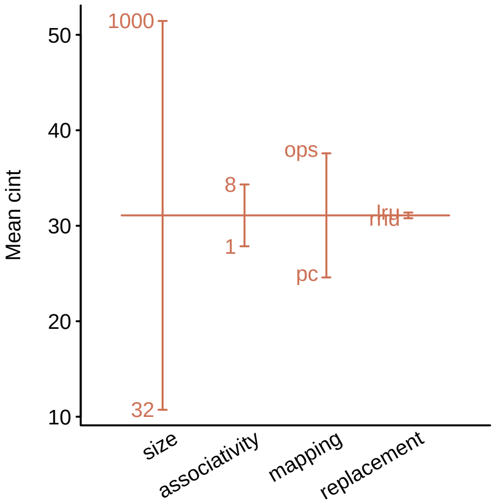


Figure 13.7: Design plot showing the impact of each configuration factor on the performance of Memo table on benchmark performance. Data from Citron et al.[358] Github–Local

13.2.6 Factorial designs

A variety of techniques are available to help visualise the results from an experiment using a factorial design. The analysis is the same for full and partial fractional designs, the only difference is the number of interactions between factors that will be available for analysis.

The study by Citron and Feitelson,[358] discussed earlier, used the SPEC CPU 2000 benchmark, and the values measured were integer floating-point performance, power consumption, processor timing, and die area of the chip.

For simplicity consider three of the factors (ignoring, for the time being, the replacement method), the results can be visualised as a cube with each of the eight vertices representing one combination of factor values; the values at each vertice of figure 13.6 are the SPEC benchmark cint performance figures.

Imagine taking two opposite faces of the above cube, say the two for `size` on the left and right going into the page, and finding the mean `cint` value for both faces, the difference between these two values is known as the *main effect* for `size`; the main effect for the other factors is similarly calculated.

A design plot is a visualisation of these main effects, with a central horizontal line showing the overall mean value of the response variable. Figure 13.7, created using the `plot.design` function, shows the impact that each factor can have on the value of `cint`, offset from the mean value.[ii]

For an example involving more changeable parameters; see Github–experiment/lee2006/lee.R.

At a finer level of granularity, an interaction plot shows the interaction between pairs of factors. Figure 13.8 shows how the mean value of `cint` varies as `size` is changed, for a given value of `mapping` (see legend).

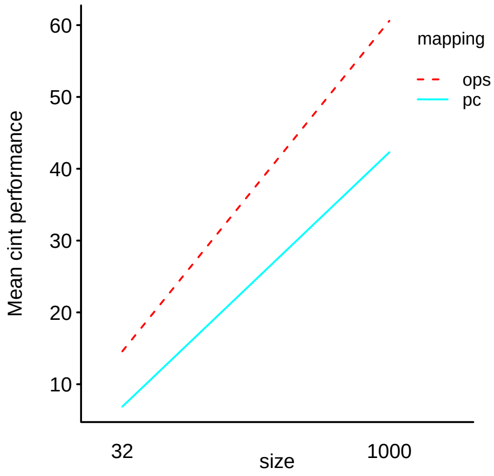


Figure 13.8: Interaction plot showing how `cint` changes with `size`, for given values of `mapping`. Data from Citron et al.[358] Github–Local

For an equation based analysis, a fitted regression model can be used to investigate the interactions between factors. For instance, the following model specifies interactions between all variable pairs; see Github–experiment/MemoPower03.R for details:

```
Memo_glm=glm(cint ~ (size+associativity+mapping)^2, data=Memo)
```

A study by Pallister, Hollis and Bennett[1419] investigated the power consumed by various embedded programs when compiled with gcc using various command lines parameters. The design used contained partial aliases, which many plotting functions cannot handle; the `halfnormal` function, from the `DoE.base` package, has an option to orthogonalize the design; see Github–experiment/pallister/gcc-power.R.

Plackett and Burman designs do not contain enough information to fit a regression model, a bespoke method has to be used, e.g., the `DanielPlot` function, in the `FrF2` package (in the plot produced, the x-axis shows effect size, the y-axis contains diagnostic information). If the data is the result of random variation (i.e., changing factor values has no effect), differences between pairs of factor averages have a (roughly) normal distribution; plotting values from a normal distribution using a normal probability scale produces a straight line. If many of the points displayed by `DanielPlot` appear to form a straight line, then the corresponding factors are likely to have had little effect on the results; any factors well off the line are of interest.

A study by Debnath, Mokbel and Lilja[456] investigated the impact of seven system configuration settings on PostgreSQL performance, on the TPC-H benchmark. High and low values were chosen for the configuration values and a Plackett and Burman design with full fold-over was used. Figure 13.9 shows the half-normal plot from the 16 runs; factors P4 and P7 do not fall on a straight line that passes close by the other factors, and they also exhibit the largest effect.

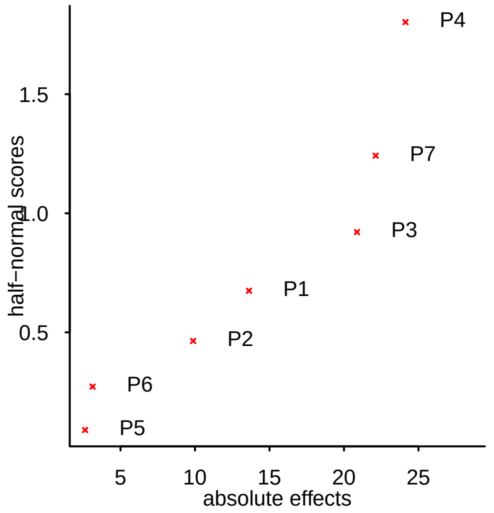


Figure 13.9: Half-normal plot of data from a Plackett and Burman design experiment. Data from Debnath et al.[456] Github–Local

13.3 Benchmarking

Benchmarking is the process of running an experiment to obtain information about the performance of some aspect of hardware and/or software. Common reasons for benchmarking, in software engineering, include comparing before/after performance and obtaining numbers to put in a report. For a more general audience, benchmarking information is often used as input to a selection process and as such often has a marketing orientation. Accurate measurements are not always necessary, showing that a system is good enough, may be good enough.

The time taken to add and multiply values was used to compare the performance of early computers.[203, 1236, 1921, 1922] Studies by Knight[1014, 1015] calculated performance based on a weighted average of instruction times, based on the kinds of instructions executed by commercial and scientific programs. Based on a study of over 300 computers available between 1944 and 1967, the rental cost for performing an operation decreased with increasing computer performance; see figure 13.10, the lines are fitted power laws with exponents between two and three; also, see Github–benchmark/EvolvingCompPerf_1963-1967.R.

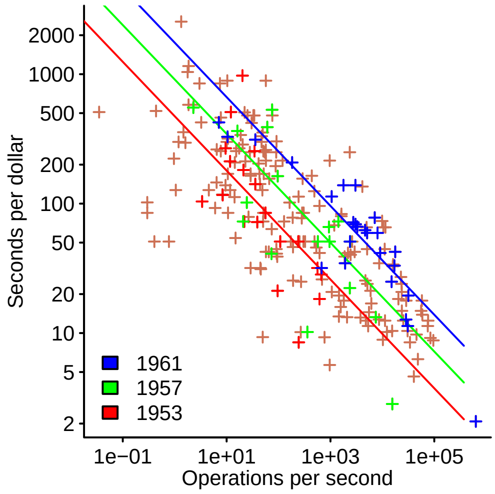


Figure 13.10: Performance and rental cost of early computers, with straight line fits for a few years. Data from Knight.[1014] Github–Local

Obtaining accurate benchmark data for many questions relating to computing platforms it may to be economically infeasible.[iii] A consequence of the continual reduction in the size

[ii] `plot.design` makes some unexpected display decisions when the explanatory variables are not factors.

[iii] Obtaining accurate benchmark results has always been an expensive and time-consuming process, but at least it was once possible to rely on devices sharing the same part number to have the same performance characteristics.

of components within microprocessors (see figure 13.11 and fig 11.51), is that individual components are now so small that variations in the fabrication process (e.g., differences in the number of atoms added or removed during the fabrication process) can noticeably change their geometry, leading to large variations in the runtime electrical characteristics of supposedly identical devices.[184]

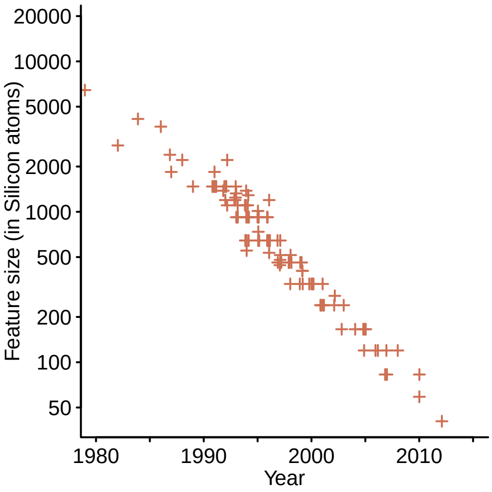


Figure 13.11: Feature size, in Silicon atoms, of microprocessors. Data from Danowitz et al.[429] Github–Local

Manufacturers offer computing systems having a range of performance characteristics; figure 13.12, lower plot, shows all published results for the integer SPEC2006 benchmark. While hardware performance has now improved to the point where for many uses it appears to be good enough, it is still possible to buy hardware that is under-powered for the job it is expected to perform (figure 13.12, upper plot, shows the orders of magnitude improvements in cost and performance of sorting over 15 years).

Benchmark results published for general consumption are sometimes little more than marketing claims. The author(s) may have reasons for wanting to create a favourable impression for one system, in preference to others, or may just have done a sloppy job (perhaps because of inexperience, incompetence or lack of resources to do a decent job). A class action suite alleged that:[665] “Intel used its enormous resources and influence in the computing industry to, in Intel’s own words, "falsely improve" the Pentium 4’s performance scores. It secretly wrote benchmark tests that would give the Pentium 4 higher scores, then released and marketed these "new" benchmarks to performance reviewers as "independent third-party" benchmarks. It paid software companies to make covert programming changes to inflate the Pentium 4’s performance scores and even disabled features on the Pentium III so that the Pentium 4’s scores would look better by comparison.”[iv] Using an established benchmark does not guarantee the results are free of vendor influence. One class action suite alleged[1590] “Samsung intentionally rigged the GS4 to operate at a higher speed when it detected certain benchmarking apps. In versions of the GS4 using the Qualcomm Snapdragon 600 processor, Samsung wrote code into the firmware (embedded software) of the GS4 to automatically and immediately drive Central Processing Unit ("CPU") voltage/frequency to their highest state, and to immediately engage all four of the processing cores of the CPU.”[v]

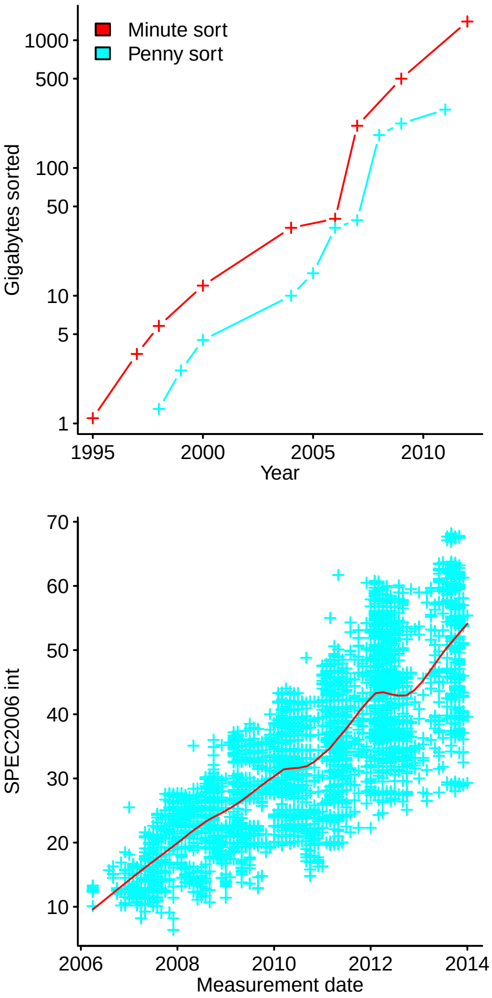


Figure 13.12: Maximum number of records sorted in 1 minute and using 1 penny’s worth of system time (upper), and SPEC2006 integer benchmark results (lower, with loess fit). Data from Gray et al[723] and SPEC.[1720] Github–Local

In some consumer goods markets, product benchmark results receive a lot of publicity, with potential customers thought to be influenced by the results achieved by similar products. A study by Shimpi and Klug,[1677] of Android benchmarks, found that some mobile phone vendors detected when a particular benchmark was being run and raised the devices thermal limits (allowing the system clock rate to run faster for longer; a 4.4% performance improvement was measured).

Researchers are happy to complain about poor benchmarking practices, but are not always willing to name names.[1855]

In published benchmark results the Devil is in the detail, or more often in the lack of detail, as illustrated by the following:

- Bailey[112] lists twelve ways in which parallel supercomputer benchmarks have been written in a way likely to mislead readers, including: quoting 32-bit, not 64-bit results, quoting figures for the inner kernel of the computation, as if they applied to the complete application, and comparing sequential code against parallelized code.
- Citron[357] analysed the ways in which many research papers using the SPEC CPU2000 suite have produced misleading results, by only using a subset of the benchmark programs (of 115 papers surveyed, 23 used the whole suite). In one case a reported speed up of 1.42 is reduced to 1.16 when the whole suite is included in the analysis (reduction from 1.43 to 1.13 in another and from 1.76 to 1.15 in a third).

The primary purpose of this section is to highlight the many sources of variability present in modern computing systems. The available evidence suggests that large variations in benchmark results are now the norm. Large variations in measured performance do not prevent accurate results being obtained, the impact is to increase the time and money needed (i.e., it is simply a case of making enough measurements). Advice on how to perform benchmarks is available elsewhere.[577, 750]

People interested in consistent performance will want to minimise the variation in benchmark results (which did occur for some programs), while those interested in actual benchmark performance will be interested that significant changes in the mean occurred for some programs.

[iv] The class action was settled[1763] with Intel agreeing to pay $15 to Pentium 4 purchasers, $4 million to a non-profit entity and an amount not to exceed $16.45 million to the lawyers who brought the suit.

[v] The class action was settled[1590] with Samsung agreeing to pay $2.55 million, with each member of the class action estimated to receive $38.38.

When comparing different systems, benchmark performance may be normalised to produce a relative performance ranking. The geometric mean needs to be used when comparing normalized values, otherwise the results can be inconsistent.

	R	M	Z	R/M	M/R	R/Z	Z/R	M/Z	Z/M
E	417.00	244.00	134.00	1.71	0.59	3.11	0.32	1.82	0.55
F	83.00	70.00	70.00	1.19	0.84	1.19	0.84	1.00	1.00
H	66.00	153.00	135.00	0.43	2.32	0.49	2.05	1.13	0.88
I	39449.00	33527.00	66000.00	1.18	0.85	0.60	1.67	0.51	1.97
K	772.00	368.00	369.00	2.10	0.48	2.09	0.48	1.00	1.00
Arithmetic	8157.40	6872.40	13341.60	1.32	1.01	1.50	1.07	1.09	1.08
Geometric	586.79	503.13	498.68	1.17	0.86	1.18	0.85	1.01	0.99

Table 13.1: Benchmark results for three processors (R, M, Z), running five benchmarks (E thru K), with normalisation using different processors, along with arithmetic and geometric means. Data from Fleming et al.[607] Github–Local

Table 13.1 shows the results of five benchmark programs (E to K) from three systems (i.e., columns R, M and Z); two other sets of columns list normalised values, with different processors used as the reference. The bottom two rows list the arithmetic and geometric mean of the columns; note: for the arithmetic mean, both ratios R/M and M/R are greater than one, while for the geometric mean one ratio is less than one (the same pattern is occurs for the other ratios). A ranking based on the arithmetic mean depends on the processor used as the base for normalization, while the geometric mean produces a consistent ranking; see section 10.3.3.

13.3.1 Following the herd

When choosing a benchmark, there is a lot to be said for doing what everybody else does, advantages include:

- can significantly reduce the cost and time needed to obtain benchmark data,
- an established benchmark is likely to be usable out-of-the-box. It takes time for a benchmark to become established; an analysis[1442] of one Java source code corpora was able to build 86 of the 106 Java systems in the corpus, with 56 of these having to be patched to get them to build,
- it is easier to sell the results to audiences, when the benchmark used is known to them.

The disadvantage of following the herd is that there may be fitness-for-purpose issues associated with using the benchmark, i.e., herd behavior is adapted to environments that may be substantially different from the environment in which the system is intended operate. For instance, the SPEC benchmark is often used to compare compiler performance, but SPEC's intent is for it to be used for benchmarking processor performance.

Commonly used benchmarks suffer from vendors tuning their products to perform well on the known characteristics of the benchmark. The SPEC benchmark has been used over many years for compiler benchmarking and compiler vendors often use it in-house for performance regression testing.

13.3.2 Variability in today's computing systems

In the good old days, computer performance tended to be relatively consistent across identical, but physically different, components, i.e., the same model of cpu or memory chip. Also, software tended to have relatively few options that could significantly alter its performance characteristics.

Modern hardware may contain components that are fabricated using handfuls of atoms, with process variations, of an atom or two here and there, producing surprisingly different performance characteristics,[1279] but externally looking like identical devices. Further reductions, in the number of atoms used to fabricate devices, will lead to greater variations in the final product. Today's consumer of benchmark results has to chose between:

- accepting a wide margin of error,
- executing a benchmark very many times, to ensure the sample size is large enough to achieve the desired statistical confidence interval for the results.

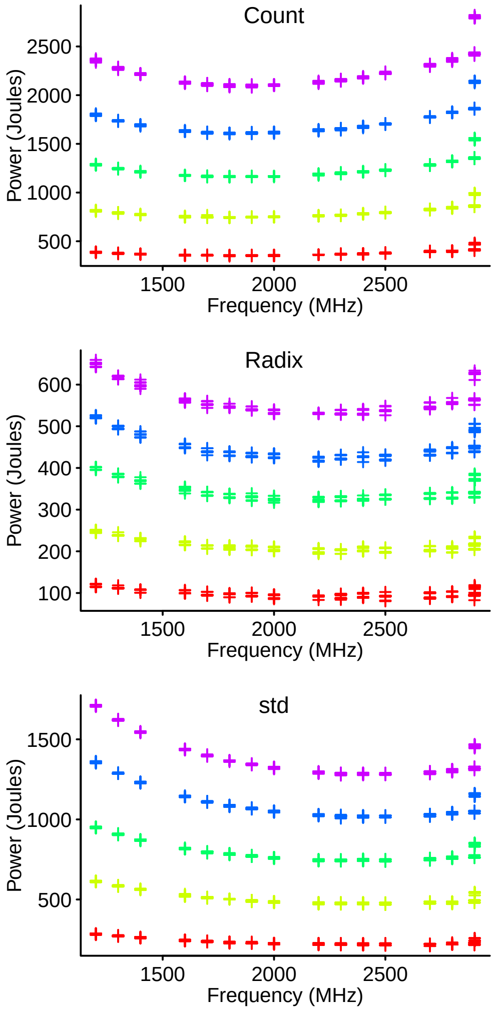


Figure 13.13: Total system power consumed when sorting 10, 20, 30, 40, 50 million integers (colored pluses) using three techniques running on the same processor at different clock frequencies. Data from Götz et al.[710] Github–Local

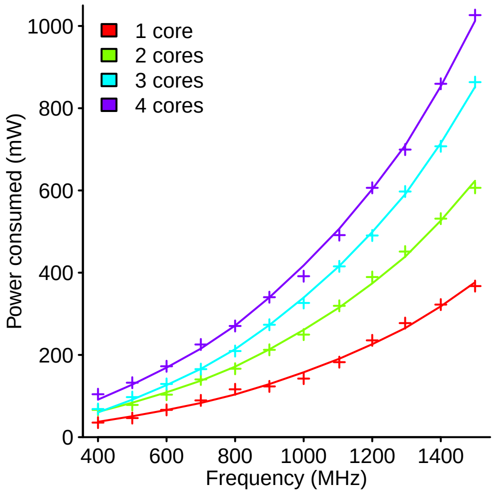


Figure 13.14: Power consumed by an Exynos-7420 A53 processor at various frequencies, and one to four cores under load, with fitted regression lines. Data kindly provided by Frumusanu.[625] Github–Local

Both approaches require checking that a wide range of, possible unknown, factors are controlled for, by those running the benchmark.

Intrinsic variability in system performance impacts development teams that regularly monitor the performance of their products during ongoing development. For instance, Mozilla regularly measures the performance of the latest checked-in version of Firefox source code, if an update results in a performance decrease exceeding a predefined limit, the update is rolled back. Successful implementation of such a policy requires careful control of external factors that could impact performance.

Performance variation has to be addressed from a system wide perspective,[vi] hardware/-software interaction can have a significant performance impact and there are often multiple, independent, sources of variation. At the systems level differences in component characteristics (e.g., differences in system clock frequency drift in multiprocessor systems[865]) can interact to produce emergent effects.

DVFS (Dynamic Voltage and Frequency Scaling) provides an example of how the complexities of system component interactions make it difficult to reliably predict performance. As its name suggests, DVFS allows processor voltage and frequency to be changed during program execution; an analysis of system power consumption[1978] concludes that total power consumed, executing a program from start to finish, is minimised by running the processor as fast as possible (assuming there is no waiting for user input).

A study by Götz, Ilsche, Cardoso, Spillner, Aßmann, Nagel and Schill[710] investigated how the total system power consumed by implementations of various algorithms varied with cpu clock frequency, with the intent of finding the frequency which minimised power consumption.

Figure 13.13 shows that total power consumption does not always decline with frequency, there is a frequency below the maximum that minimises power consumed.[452] The power minimisation frequency depends on the implementation of the sorting algorithm, with the difference between minimum and maximum depending on the number of items being sorted. Predicting the power consumed[196] by a program is a non-trivial problem.

Programming languages are starting to support constructs that provide developers with options for dealing with power consumption issues.[1611]

13.3.2.1 Hardware variation

This section outlines some evidence for large variations in hardware component performance. Much of the data used in the analysis was obtained using programs executing on components manufactured five to ten years before this book was published; variability has likely increased in the subsequent years. The hardware components covered include:

- CPU: performance, power consumption and instruction counts,
- main storage: hard disc performance and power consumption,
- memory: performance and power consumption,

When computing devices are connected to the mains power supply, there is rarely any need to be concerned about the characteristics of the supply. Batteries have characteristics that can affect the performance of devices connected to them, such as the level of power delivery being dependent on the current charge state and power draw frequency characteristics. In a mobile computing environment, power consumption can be just as important as runtime performance, if not more so; there are limits to the amount of electrochemical energy that can be stored.[1203] Peltonen et al[1445] is a public dataset of power consumption on 149,788 mobile devices, containing 2,535 different Android models; Jongerden[932] analyses various models of battery powered systems and Buchmann[265] covers rechargeable batteries in detail.

In mobile devices, a large percentage of power is consumed by the display; optimization of display intensity and choice of color,[1737] while an app is running, is not discussed here.

CMOS (complementary metal-oxide-semiconductor) is the dominant technology used in the fabrication of the chips contained in computing devices; until a so-called beyond-CMOS device[1363] technology becomes commercially viable, this section only considers the characteristics of CMOS devices.

[vi] The following two sections separately discuss performance variation whose root cause in hardware or software; this is for simplicity of presentation.

CPU: the processors executing code are often considered to be interchangeable with any other, mass-produced, etched slices of silicon stamped with the same model number; while never exactly true, deviations from this interchangeability assumption were once small enough to only be of interest within specialised niches, e.g., hardware modders interested in running systems beyond rated limits.

The micro-architecture of modern processors has become so complicated that apparently minor changes to an instruction sequence can have a major impact on performance;[864] a trivial change to the source code, or the use of a different compiler flag may be enough.

Power consumption and clock frequency are directly connected; increasing clock frequency increases power consumption (a good approximation for processor power consumption is $P = \alpha F V^2 + I_0 V$, where: α is a device dependent constant, F is clock frequency, V is voltage supplied to the cpu,[vii] and I_0 is leakage current). Processors clocked at the same frequency execute instructions at the same rate. However, variations in the number of atoms implementing internal circuitry produces variations in power consumption. Some processors reach their maximum operating temperature more quickly than others; to prevent device destruction through overheating, power consumption is reduced by reducing the clock rate. Different processors have different sustained performance rates because of differences in their power consumption characteristics. Vogeleer[451] discusses the modeling of low level temperature/power relationships for the kind of processors used to run applications.

A study by Frumusanu[625] measured the power and voltage, at various frequencies, of an Exynos-7420 A53 processor idling and at load. Figure 13.14 shows measured power consumption, involving one to four cores under load, at various frequencies and a fitted regression model.

Any benchmark made using a single instance of a processor is a sample drawn from a population that could vary by something like 5-10% or more when executing code and several hundred percent when idling. The extent to which results based on this minimum sample size is of practical use will depend on the questions being asked. If the power consumption characteristics of the population of a particular CPU is required, then it is necessary to benchmark a sample containing an appropriate number of *identical* processors. Methodologies for benchmarking power consumption[1721] require detailed attention to many issues.

A study by Wanner, Apte, Balani, Gupta and Srivastava[1905] measured the power consumed by 10 separate Amtel SAM3U microcontrollers at various ambient temperatures. Figure 13.15 shows a 5-to-1 difference, between supposedly identical processors, in power consumption when in sleep-mode (upper plot), and around 5% difference when operating at 4MHz.

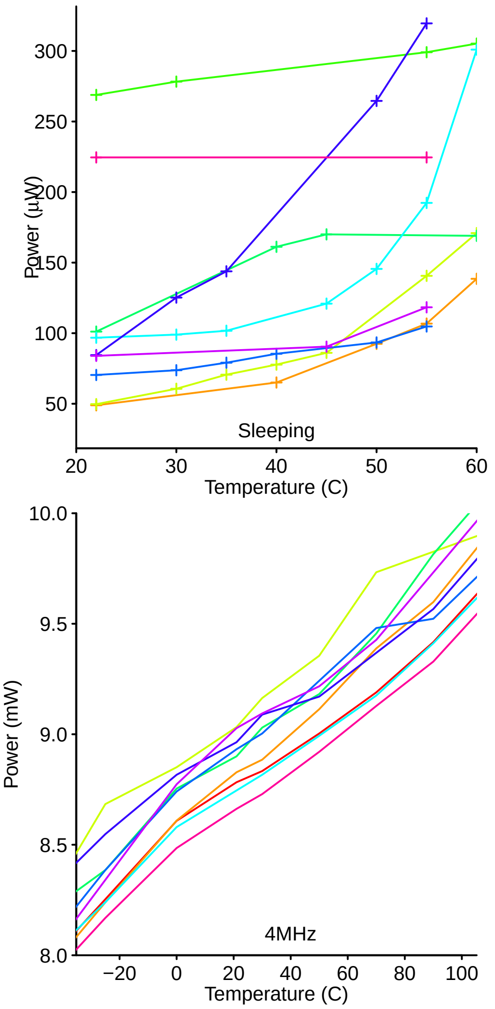


Figure 13.15: Power consumed by 10 Amtel SAM3U microcontrollers at various temperatures when sleeping or running. Data from Wanner et al.[1905] Github–Local

Section 11.6 discusses the building of mixed-effects models for power variations of the Intel Core processor.

A study by Marathe, Zhang, Blanks, Kumbhare, Abdulla and Rountree[1189] investigated the variation in performance of 2,386 Intel Sandy Bridge XEON processors while operating under a running average power limit (RAPL). Figure 13.16 shows the time taken by 2,386 processors to complete the Embarrassingly parallel benchmark and their clock frequency, with a RAPL of 65 Watts.

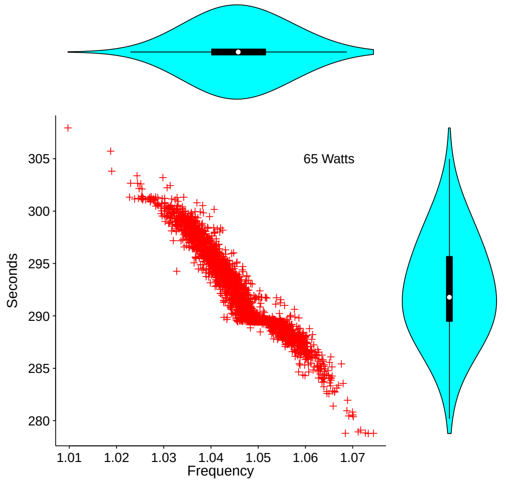


Figure 13.16: Time taken to execute the EP benchmark and clock frequency of 2,386 Intel processors, with a RAPL of 65 Watts. Data kindly provided by Rountree.[1189] Github–Local

How accurate are power consumption measurements? These measurements are often implemented by periodically sampling the voltage across a known resistance. A study by Saborido, Arnaoudova, Beltrame, Khomh and Antoniol[1601] investigated the measurement error introduced by different sampling rates, on mobile devices. Figure 13.17 shows the power spectrum of the Botanica App, executing on a BeagleBone Black running Android 4.2.2, sampled at 500K per second. By using a very high sampling rate, it is possible to see the noticeable peak in power consumed by very short-lived events, something that low frequency sampling would not detect (the paper lists error estimates for lower sampling rates).

In theory, counting the instruction executed by a program is a means of obtaining, power independent, answers to questions about comparative program performance. Some processors include hardware support for counting the number of operations performed, e.g., instruction opcodes executed and cache misses; however, the purpose of these counters is

[vii] On some processors there is a linear relationship between voltage and frequency,[451] i.e., $P = \beta V^3 + I_0 V$, or $P = \gamma F^3 + I_0 V$.

to help manufacturers debug their processors, not provide end-user functionality. Consequently, counter values are not guaranteed to be consistent across variants of processors within the same family[1914] and fixing faults in the counting hardware does not have a high priority (e.g., counting some instructions twice or not at all; Weaver and Dongarra[1914] found that in most cases the differences were a fraction of a percent of the total count, but for some kinds of instructions, such as floating-point, the counts were substantially different). A study by Weaver and McKee[1915] found that it was possible to adjust for the known faults in the hardware counters; see Github–benchmark/iiswc2008-i686.R.

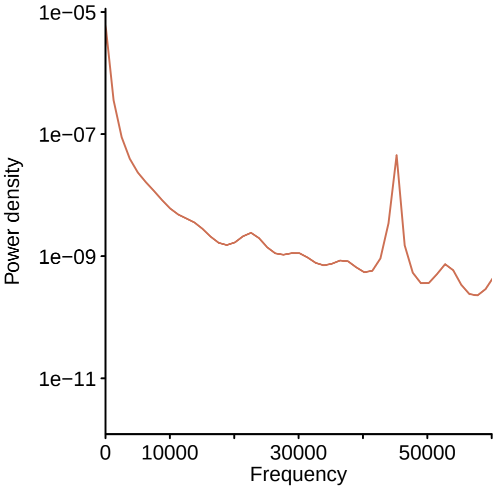


Figure 13.17: Power spectrum of the electrical power consumed by the Botanica App executing on a BeagleBone Black running Android 4.2.2. Data from Saborido et al.[1601] Github–Local

How are calls into operating system routines, which may execute at higher privilege levels, counted? Also, the execution time of some instructions may depend on other instructions executed at around the same time (modern processors have multiple functional units and allocate resources based on the instructions currently in the pipeline), the time taken to execute other instructions can be very unpredictable (e.g., time taken to load a value from memory depends on the current contents of the cache and other outstanding load requests). A study by Melhus and Jensen[1243] showed that address aliasing, of objects in memory, could have a huge impact on the relative values of some hardware performance counters.

Hardware counters need not be immune to *observer effects*; in particular, the values returned can depend on the number of different hardware counters being collected.[1324]

Main storage: Traditionally main storage has meant hard disks (sometimes backed-up with tape), but solid state devices (SSD) are rapidly growing in capacity[1779] and use; for extremely large capacity magnetic tape is used: this niche use is not discussed here.

Data is read/written to a hard disk by moving magnetic sensors across a rotating surface. These spatial movements create a correlation between successive operations, e.g., the time taken to perform the second read will depend on its location on the disk relative to the first. Disks spin at a constant rate, and in the same time interval more data can be read from the area swept out near the outer edge of a platter than from one near the spindle (the staircase effect in the upper plot in figure 13.18 is a result of zoning). This location dependent performance characteristic makes disk benchmark performance dependent on the history of the data that has been added/deleted.

Further correlations are created by data buffering, by the operating system and the device itself, and access requests being reordered to optimise overall throughput.

Storage farms organise files so that those most likely to be accessed are stored on the outer tracks, while files less likely to be accessed are stored on the inner tracks. A growing percentage of disks are used in data centers and at some point manufacturers may decide to concentrate on designing drives for this market.[249]

The continuing increase in the number of bits that can be stored within the same area of rotating rust has been achieved by reducing the size of the magnetic domain used to store a bit. Like silicon wafer production, variations in the fabrication process of disc platters can now result in large differences in the performance of supposedly identical drives.

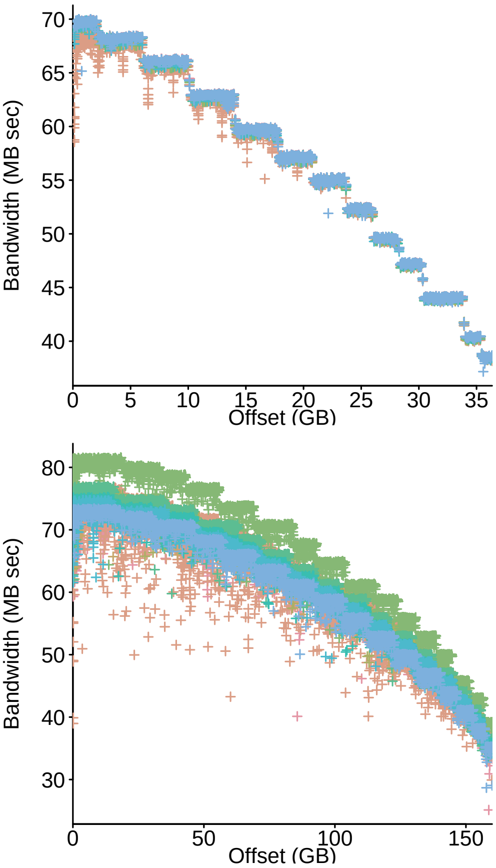


Figure 13.18: Read bandwidth at various offsets for new disks sold in 2002 (upper) and 2006 (lower). Data kindly provided by Krevat.[1032] Github–Local

A study by Krevat, Tucek and Ganger[1032] measured the performance of disk drives originally sold in 2002, 2006, 2008 and 2009. Figure 13.18, upper plot, shows the read bandwidth of nine disks from 2002, each displayed using a different color and there is little variation between different disks (fitting a regression model finds that disk identity is not a significant predictor of performance, p-values around 0.2). The lower plot shows the read bandwidth of nine disks from 2006, each displayed using a different color; the visibility of different colors shows the variation between different disks (fitting a regression models finds that disk identity is a significant component of performance prediction, p-values around 10^{-16}).

To increase recording density, drive manufacturers are now using Shingled Magnetic Recording (SMR), where tracks overlap like rows of shingles on a roof. Singled discs have very different performance characteristics,[14] but little data is publicly available at the time of writing.

SSDs are sufficiently new that little performance data is publicly available at the time of writing.

A study by Kim[986] ran eight different benchmarks on SSD cards from nine different vendors. The range of performance values was different for both vendors and benchmark. Building a regression model, using normalised benchmark scores, finds that one vendor's products have a sufficiently consistent performance that they can be included in a model (these products appear to have the best performance, the other vendors appear to have the same performance); see Github–benchmark/hyojun/hyojun.R.

Memory: Memory chips tend to be thought about in terms of their capacity and not their performance (such as, read/write delays or power consumption). Performance is governed by access rate and by the number of bytes transferred per access, with accesses usually made via some form of memory control chip (the capabilities of this controller have a significant impact on performance). Many motherboards provide options to select memory chip timing characteristics.

A study by Bircher[196] investigated the power consumed by the various hardware of a server, while it executed the SPEC CPU2006 benchmark. Figure 13.19, upper plot, breaks down average power by CPU (red) and memory (blue), while the lower plot breaks the power down by the major subcomponents of the server.

There is often a performance hierarchy for memory, with on (cpu) chip cache providing faster access to frequently used data. The interaction between the size of the various memory caches, and an algorithm's use of storage, can result in performance characteristics that change as the size of the objects processed changes.

A study by Khuong and Morin[983] measured the performance of several search algorithms, when operating on arrays of various sizes (items were stored appropriately in the array, for the algorithm used). Figure 13.20 shows the time taken by different algorithms to find an item in an array, for arrays of various sizes; the grey lines show the total size of the processor L1, L2 and L3 caches.

The following are some examples of memory chip characteristics that have been found to noticeably fluctuate:

- a study by Gottscho, Kagalwalla and Gupta[709] measured power consumption variability of 13 DIMMs, of the same model of 1G DRAM from four vendors. The variation about the mean, at one standard deviation, was 5% for read operations, 9% for write and 7% for idling; see Github–benchmark/J20_paper.R:
- a study by Gottscho[708] measured the power consumption of 22 DDR3 DRAMs, manufactured in 2010 and 2011, from four vendors. Read operations consumed around 60% of the power needed for write operations, with idle consuming around 40%; the standard deviation varied from 10% to 20%. The power consumed also varied with value being read/written, e.g., writing 1 to storage containing a 0 required 25% more power than writing a 0 over a 1; see Github–benchmark/MSTR10-DIMM.R for data.
- a study by Schöne, Hackenberg and Molka[1628] found that memory bandwidth was reduced by up to 60%, as the frequency of the cpu was reduced, that memory performance characteristics varied between consecutive generations of Intel processors and between server and desktop parts,

The variability of memory chip performance is likely to increase, as vendors further reduce power consumption and improve performance by lengthening DRAM refresh times; optimising each computer by tuning it to the unique characteristics of the particular chips present in each system.[1092]

Chandrasekar[314] provides a detailed discussion of DRAM power issues, including code for a tool to obtain detailed information about the memory chips installed on a system.

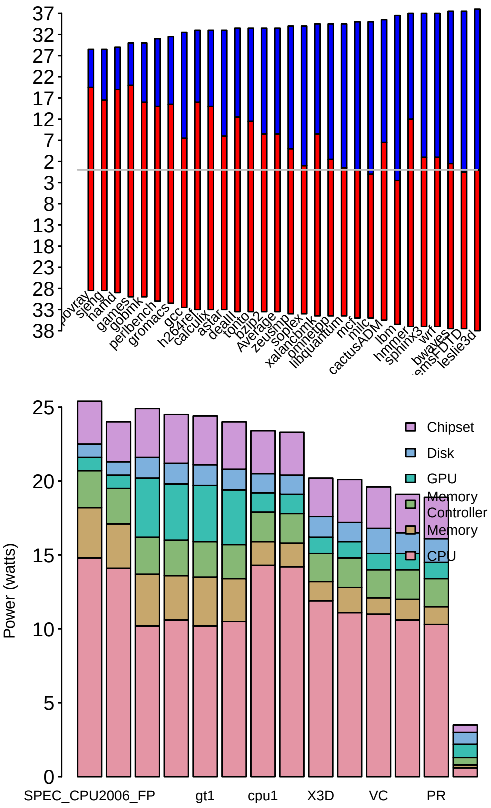


Figure 13.19: Average power consumed by one server's CPU (four Pentium 4 Xeons; red) and memory (8 GB PC133 DIMMs; blue) running the SPEC CPU2006 benchmark (upper) and breakdown by system component when executing various programs. Data from Bircher.[196] Github–Local

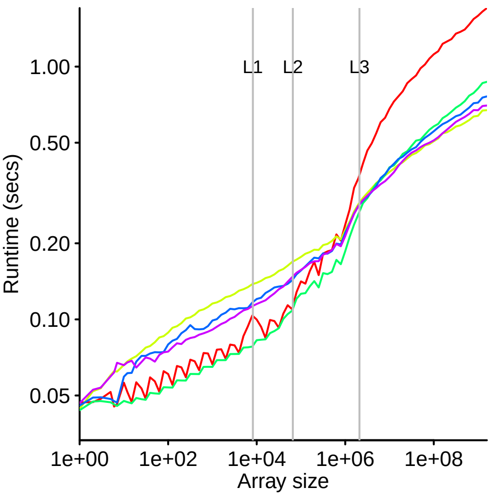


Figure 13.20: Time taken to find a unique item in arrays of various size, containing distinct items, using various search algorithms; grey lines are L1, L2 and L3 processor cache sizes. Data from Khuong et al.[983] Github–Local

13.3.2.2 Software variation

This section outlines some of the evidence for large variations in software performance, briefly covering the following software components and processes:

- The environment: interaction with the environment, file system, support libraries and aging,
- Configurations,
- Creating an executable: compiler optimization and link order,
- Tools.

The environment: Programs execute within an environment that often contains a complicated ensemble of interconnecting processes and services that cannot be treated as independent standalone components. One consequence of this complexity,[914] and interconnectedness, is that the order in which processes are initiated during system startup can have a noticeable impact on system performance.

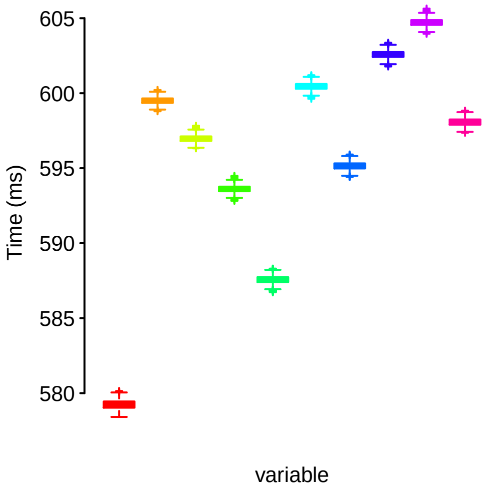


Figure 13.21: FFT benchmark executed 2,048 times followed by system reboot, repeated 10 times. Data kindly provided by from.[951] Github–Local

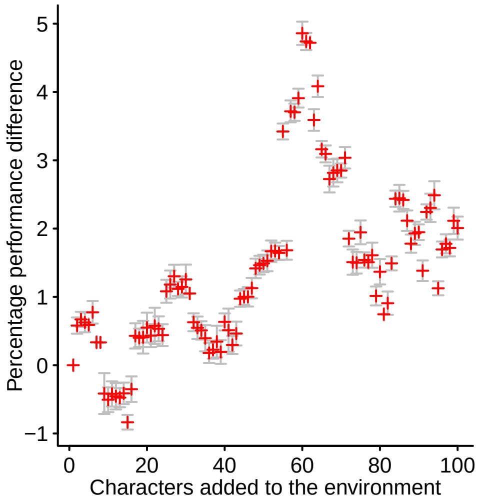


Figure 13.22: Percentage change, relative to no environment variables, in perlbench performance as characters are added to the environment. Data extracted from Mytkowicz et al.[1323] Github–Local

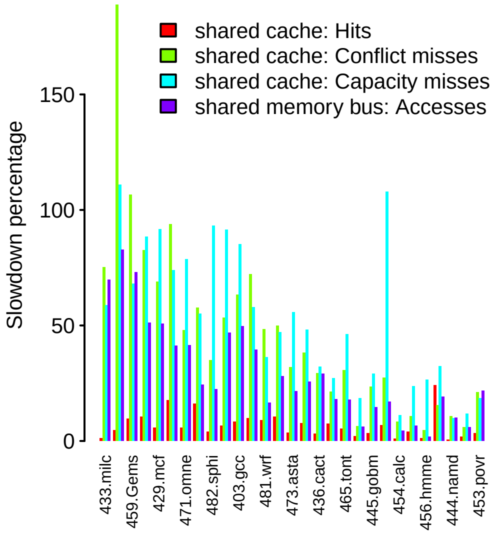


Figure 13.23: Changes in SPEC CPU2006 performance caused by cache and memory bus contention, for one dual processor Intel Xeon E5345 system. Data kindly provided by Babka.[97] Github–Local

The impact of a system's prior history, on program performance, is seen in a study by Kalibera, Bulej and Tůma,[951] who measured the execution time of multiple runs of various programs. Figure 13.21 shows 10 iterations of the procedure: reboot computer and make 2,048 performance measurements. The results show performance variation after each reboot is around 0.1%, but rebooting can cause a shift of 3% in the average performance (the ordering of processes executed during system startup varies across reboots, due to small changes in the time taken to execute the many small scripts that are invoked during startup; execution ordering affects placement of data in memory, which can have an impact on performance). A later study[827] found that the non-determinism of initial program execution, in this case, could be reduced by having the operating system use cache-aware page allocation.

Environmental interactions are not always obvious. A study by Mytkowicz, Diwan, Hauswirth and Sweeney[1323] increased the number of bytes occupied by a Linux environment variable between runs of the `Perlbench` program. The results from each of 15 executions were recorded, an environment variable increased in size by one character, and the procedure repeated 100 times. Figure 13.22 shows the percentage change in performance, relative to the environment variable containing zero characters, at each size of environment variable, along with 95% confidence intervals of the mean of each 15 runs.

Incremental operating system updates can produce a change in program performance. A study by Flater[605] compared the performance of cpu intensive and I/O bound programs on two different versions of Slackware, running on the same hardware (versions 14.0 and 14.1, using Linux kernels 3.12.6 and 3.14.3 respectively). The results show consistent differences in performances of up to 1.5% (rebooting did not have any significant impact on performance).

Many systems allow multiple programs to share system resources, by executing at the same time. Sharing becomes a performance bottleneck when one program cannot immediately access resources when it requests them; access to memory is a common resource contention issue on multi-processing systems. A study by Babka[97] investigated the performance of multicore processors having a shared cache. Figure 13.23 shows changes in SPEC CPU2006 performance caused by cache and memory bus resource contention, on a dual processor Intel Xeon E5345 system.

A study by Mazouz[1210] investigated the performance of the SPEC OpenMP 2001 programs, compiled using gcc 4.3.2 and icc 11.0, running on multicore devices. It is possible for a program's code to execute on a different core after every context switch. Allowing the operating system to select the core to continue program execution is good for system level load balancing, but can reduce the performance of individual programs because recently accessed data is less likely to be present in the cache of any newly selected core. *Thread affinity* is the process of assigning each thread to a subset of cores, with the intent of improving data locality, i.e., recently accessed data is more likely to available in accessible caches.

Figure 13.24 shows the time taken to execute one program in 2, 4, and 6 threads, with thread affinity set to compact (threads share an L2 cache), no affinity (allow the OS to assign threads to cores) and scatter (distribute the threads evenly over all cores), each repeated 35 times.

Configuring the system being benchmarked to only run one program at a time solves some, but not all, cache contention issues. Walking through memory, in a loop, may result in a small subset of the available cache storage being used (main memory is mapped to a much smaller cache memory, which means that many main memory addresses are mapped to the same cache address). Figure 13.25, from a study by Babka and Tůma,[98] shows the effect of walking through memory using three different fixed width strides; for 32 and 64 byte strides accesses to even cache lines is faster than odd lines, with the pattern reversed for a 128 byte stride.

Operating systems generally have background processes that spend most of their time idling, but wake up every now and again. When a background processes wakes up, it will consume system resources and can have an impact on the performance reported by a benchmark, i.e., background processes are a source of variation.

A study by Larres[1076] investigated how the performance of one version of Firefox changed as various operating system features were disabled (the intent being to reduce the likelihood that external factors added noise to the result). The operating system features modified were: 1) every process that was not necessary was terminated, 2) address-space randomization was disabled, 3) the Firefox process was bound exclusively to one cpu, and 4) the Firefox binary was copied to and executed from a RAMDISK.

Every program in the Talos benchmark (the performance testing framework used by Mozilla) was run 30 times. Figure 13.26 shows the performance of various programs running in original and stabilised (i.e., low-noise) configurations.

File systems: These provide the housekeeping structure for keeping track of information on a storage device. The traditional view of a file, as a leaf in a directory tree, has become blurred, with many file system managers now treating compressed archived files (e.g., zip files) as-if they had a directory structure that can be traversed; Microsoft's .doc format contains a FAT (File Allocation Table, just like a mounted Windows file system) that can refer to contents that may exist outside the *file*, other vendor applications can be more complicated.[774]

A study by Zhou, Huang, Li and Wang[2002] investigated the performance interplay between file systems and Solid State Disks (SSD), by running a file-server benchmark on a Kingston MLC 60 GB SSD. Four commonly used Linux filesystems (ext2, ext3, reiserfs and xfs) were mounted in turn using various options, e.g., various block sizes, noatime, etc.

Figure 13.27 shows the number of operations per second for a file-server benchmark (see paper and data for other benchmarks). A linear regression model involving the filesystem and mount options is a poor fit to the data; see Github–benchmark/filesystem-SSD.R.

A study by Sehgal, Tarasov and Zadok[1647] compared the power used when four commonly used filesystems were mounted in various ways, e.g., fixed vs. variable sector size, different journal modes, etc. Various server workloads running on Linux were measured; web server power consumption varied by a factor of eight, mail server by a factor of six and file and database by a factor of two.

Creating an executable: Many applications are built by translating source code to an executable binary, with the translation tools often supporting many options, e.g., gcc supports over 160 different options for controlling machine independent optimization behavior. Compiler writers strive to improve the quality of generated code, and it is to be expected that the performance of each release of a compiler will be different from the previous one; there have been around 150 released versions of gcc in its 30-year history.

A study by Makarow[1182] measured the performance of nine releases of gcc, made between 2003 and 2010, on the same computer using the same benchmark suite (SPEC2000), at optimization levels 02 and 03.

Figure 13.28 shows the percentage change in SPEC number, relative to version 4.0.4, for the 12 integer benchmark programs compiled using six different versions of gcc. SPEC has a long history of being used for compiler benchmarking, and it is possible that the versions of gcc used for this comparison have already been tuned to do well on this benchmark, meaning there is little, benchmark specific, improvement to be had in the successive versions used in this study.

The following `summary` output is from a mixed-effect model with the random effect on the intercept and slope: Github–Local

```
Linear mixed model fit by REML ['lmerMod']
Formula: value ~ gcc_version + (gcc_version | Name)
   Data: lme_02

REML criterion at convergence: 400.6

Scaled residuals:
    Min      1Q  Median      3Q     Max
-2.7256 -0.2748 -0.0683  0.3039  4.3372

Random effects:
 Groups   Name        Variance Std.Dev. Corr
 Name     (Intercept) 1192.792 34.537
          gcc_version    3.155  1.776   -1.00
 Residual                8.632  2.938
Number of obs: 72, groups:  Name, 12

Fixed effects:
            Estimate Std. Error t value
```

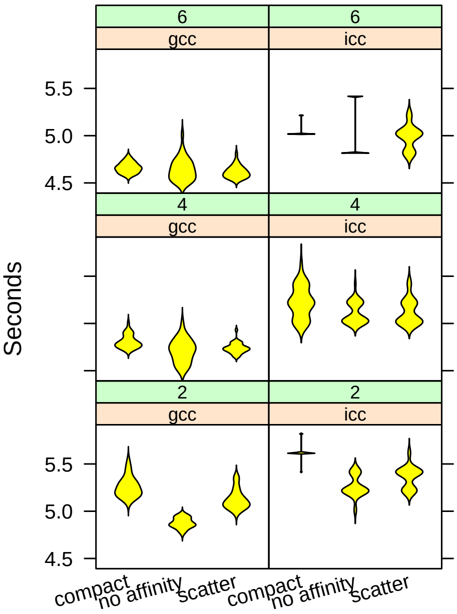


Figure 13.24: Execution time of `330.art_m`, an OpenMP benchmark program, using different compilers, number of threads and setting of thread affinity. Data kindly provided by Mazouz.[1210] Github–Local

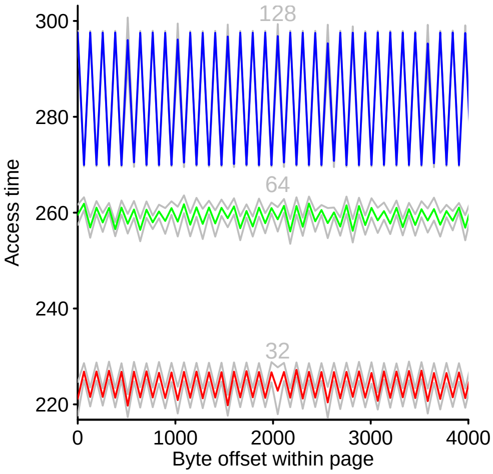


Figure 13.25: Access times when walking through memory using three fixed stride patterns (i.e., 32, 64 and 128 bytes) on a quad-core Intel Xeon E5345; grey lines at one standard deviation. Data kindly provided by Babka.[98] Github–Local

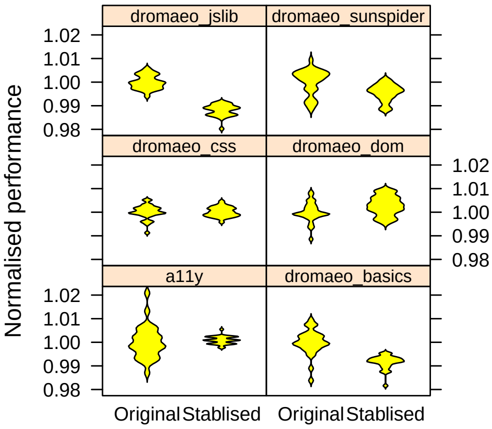


Figure 13.26: Performance variation of programs from the Talos benchmark run on original OS and a *stabilised* OS. Data from Larres.[1076] Github–Local

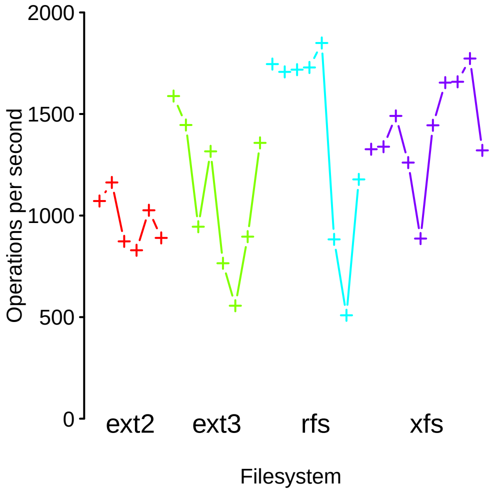


Figure 13.27: Operations per second of a file-sever mounted on one of ext2, ext3, rfs and xfs filesystems (same color for each filesystem) using various options. Data kindly supplied by Huang.[2002] Github–Local

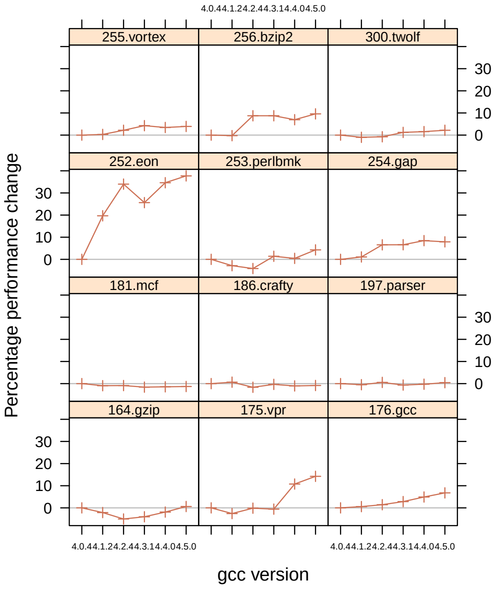


Figure 13.28: Percentage change in SPEC number, relative to version 4.0.4, for 12 programs compiled using six different versions of gcc (compiling to 64-bits with the 03 option). Data from Makarow.[1182] Github–Local

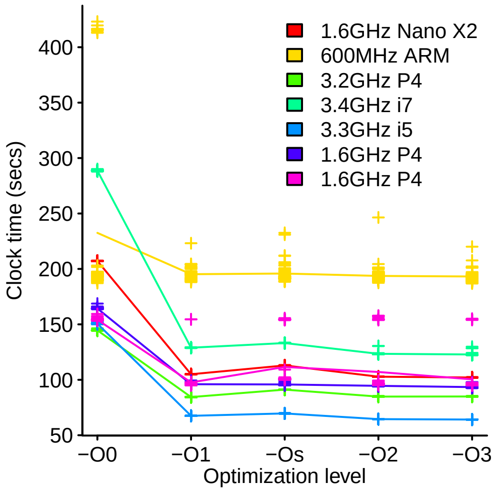


Figure 13.29: Execution time of the xy file compressor, compiled using gcc using various optimization options, running on various systems (lines are mean execution time when compiled using each option). Data kindly supplied by Petkovich.[447] Github–Local

```
(Intercept) -29.7469    11.0553  -2.691
gcc_version   1.4126     0.5513   2.562

Correlation of Fixed Effects:
            (Intr)
gcc_version -0.997
convergence code: 0
boundary (singular) fit: see ?isSingular
```

The general picture painted by the model results is of a small improvement with each gcc release, which is swamped by the size of the random effects, while the picture painted by figure 13.28, is of some releases having a large impact on some programs.

A study by de Oliveira, Petkovich, Reidemeister and Fischmeister[447] investigated the impact of compiler optimization and object module link order on program performance. Figure 13.29 shows the time taken by the xy file compression program, compiled by gcc using various optimization options, to process the Maximum Compression test set on various systems. The results show that different optimization levels have a different performance impact on different systems (the lines would be parallel if optimization level had the same impact for each system).

Compiling is the first step in the chain of introducing system variability into program performance, the next step is linking. Figure 13.30 shows execution time of Perlbench (one of the SPEC benchmark programs), on six systems, when the object files used to build the executable are linked in three different orders and with address randomization on/off. Some systems share a consistent performance pattern across link orderings, and some systems are not affected by address randomization. But there is plenty of variation across all the variables measured.

Tools: Dynamic profiling tools such a grpof work by interrupting a program at regular intervals during execution (e.g., once every 0.01 seconds) and recording the current code location (often at the granularity of a complete function). The results obtained can depend on interrupt frequency and the likelihood of being in the process of calling/returning from the profiled function.[604]

13.3.3 The cloud

Cloud computing has become a popular platform for applications that require non-trivial compute resources. The service level agreements offered by cloud providers specify minimum levels of service, e.g., Amazon's June 2013 EC2 terms specify 99.95% monthly uptime.[48] Cloud services general run virtualized instances, which means access to the real hardware may sometimes be shared. Shared hardware access causes performance to vary from one run to the next; what form might the characteristics of this variation take?

A study by Schad, Dittrich and Quiané-Ruiz[1619] submitted various benchmarks, as jobs, to Amazon's Elastic Computing Cloud (EC2), twice an hour over a 31-day period; a variety of resource usage measurements were recorded. Figure 13.31 shows one set of resource usage measurements, the Unix benchmark utility (Ubench; a cpu benchmark) running on small (upper) and large (lower) EC2 instances located both in Europe (red) and the US (green).

Both plots show more than one distinct ranges of performance. This data is an example of the variation experienced in Amazon's EC2 performance over one particular time period, and there is no reason to believe that any subsequent benchmarking will exhibit one, two, three or more distinct performance ranges.

13.3.4 End user systems

Benchmark data supplied by end-users, run on the computing systems they own, is likely to be subject to numerous known and unknown unknowns.

It may be impractical to switch off the many background processes that may be running on, for instance a user's Windows machine, which might include: Internet based toolbars, anti-virus systems and general OS housekeeping processes. PassMark Software specializes in benchmark solutions for Microsoft Windows based computers, and Wren[1960] kindly provided 10,000 memory benchmark results.

Figure 13.32 shows the results (in sorted order) from 783 systems containing an Intel Core i7-3770K processor (whose official clock speed is 3.5GHz, some users may be overclocking). This is another example (see fig 10.25) of the wide range of performance reported for apparently very similar end-user systems.

User applications can have complex internal structures and modes of operation that invalidate assumptions made by a benchmark. For instance, application data files may not be represented as a contiguous sequence of bytes, but contain internal meta-data and pointers to blocks of data in other files.[774]

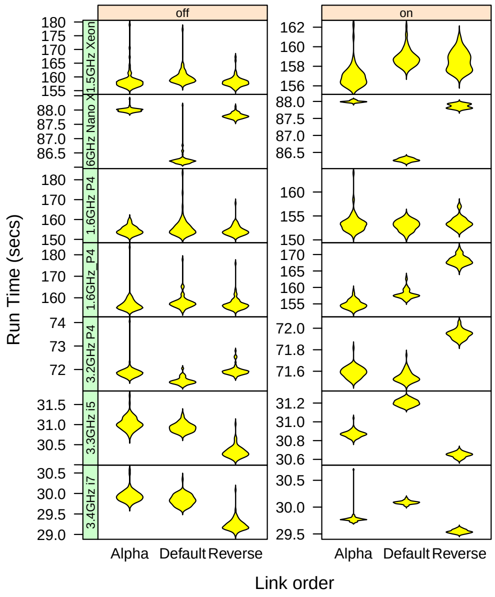


Figure 13.30: Execution time of Perlbench, part of the SPEC benchmark, on six systems, when linked in three different orders and address randomization on/off. Data kindly supplied by Reidemeister.[447] Github–Local

13.4 Surveys

This section discusses questionnaire surveys. Organizations use surveys are used to obtain information about customers and the market(s) they are targeting, e.g., characteristics of open source developers;[1574] see fig 8.13. Applications need to run reasonably well on the computers that customers currently use (see fig 8.27), and to coexist (or interoperate) with the versions of libraries and other applications installed on these computers. A lot of software engineering information only exists in the heads' of the people who build software systems, and this information can only be obtained by asking these people questions and analysing their answers.

The `survey` package supports the analysis of samples obtained via surveys.

The characteristics of data encountered in survey samples include:[415]

- missing data: people don't answer all the questions or stop answering after some point,
- misleading answers: giving answers that show those involved in a better light, such as job adverts listing trendy topics and languages to attract more applicants,
- spatial information: how subjects are distributed geographically,

Studies have found[511] that self-assessment of skills and character have a tenuous to modest relationship with actual performance and behavior. The correlation between self-ratings of skill and actual performance in many domains is moderate to meager.

Several studies by your author[920, 921, 923] included a component that asked developers about how many lines of code they had read and written during their professional career.

This question requires a lot of thought to answer, and there are many ways of adding up the numbers. Does reading the same line twice count as two lines, or one line unless the developer involved had forgotten reading it? How much does visually searching a screen of code (e.g., for a particular identifier) count towards lines read? Counting the number of lines in the programs written by a developer is likely to underestimate the number of lines they have written; a line of code may be written and then deleted, an existing line may be modified slightly.

Figure 13.33 shows the number of lines of code that 101 professional developers estimate they have written. While an exponential model fits the data, the variance explained is small.

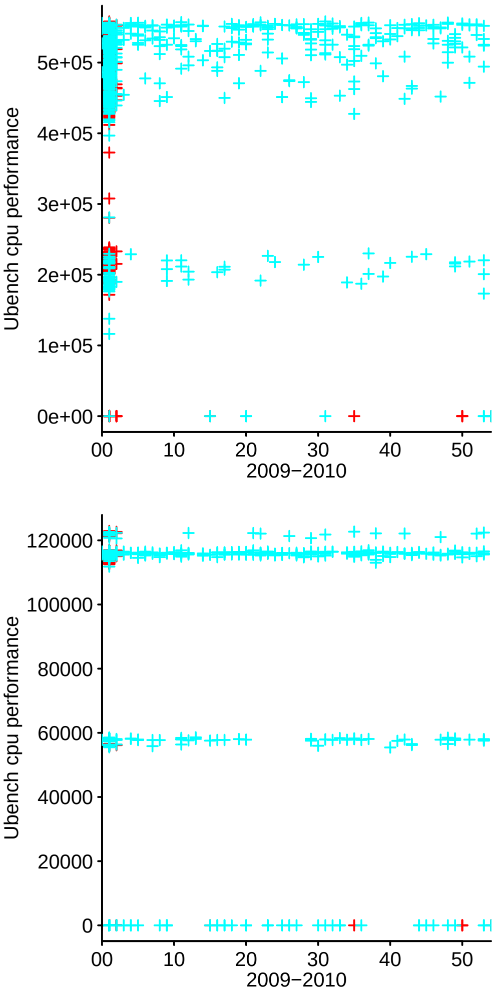


Figure 13.31: Ubench cpu performance on small (upper) and large (lower) EC2 instances, Europe in red and US in green. Data kindly provided by Dittrich.[1619] Github–Local

A survey of the knowledge, or skill, of members of a population requires subjects to provide correct answers to questions.

Item response theory (IRT) deals with the design, analysis, and scoring of tests and questionnaires. The `ltm` package (latent trait models) supports the analysis of item response data.

What is the probability that a subject, m, will give the correct answer to the i^{th} question, x_{mi}, when the subject has a knowledge/skill level of z_m? The answer given by IRT[152] is:

$$P(x_{mi} = 1|z_m) = c_i + (1 - c_i)g(\alpha_i \times (z_m - \beta_m))$$

where: c_i is the probability that a subject will guess the correct answer, α_i is a measure of how well the question discriminates between subjects having a low/high level, β_i the question difficulty, and $g(.)$ a link function (often the logit function; the default used by the function in the `ltm` package).

The Rasch model is simpler, and widely used; it contains a single parameter, β_i, and assumes there is no guessing (i.e., $c_i = 0$), and questions share the same ability to discriminate between subjects at different levels (i.e., $\alpha_i = 1$); the logit function is used as the link function, and the equation is:

$$P(x_{mi} = 1|z_m) = \frac{e^{z_m - \beta_m}}{1 + e^{z_m - \beta_m}}$$

The `ltm` package supports the fitting of Item response models having one (the Rasch model), two (α_i is included) and three (all parameters are included) parameter models; the logit function is the default link function.

A study by Dietrich, Jezek and Brada[490] investigated one aspect of developer knowledge of a language: knowledge of Java type compatibility. The yes/no answers to 22 questions provided by the 184 professional developers can be fitted to an IRT model.

The following code fits a Rasch model (single parameter), and a two parameter model using the `ltm` function (`z1` and `z2` are dummy names used to denote each factor); a three parameter model fails to be fitted by the `tpm` function.

```
library("ltm")

corr_mod1=ltm(corr_df ~ z1)

plot(corr_mod1)

corr_mod2=ltm(corr_df ~ z1+z2)

summary(corr_mod2)

t=tpm(corr_df) # fails to fit
```

Figure 13.34 shows: the probability that a subject having a given level of ability will correctly answer each question.

Section 12.4.2 discusses the analysis of ranked items, i.e., placed in a preferred order.

The results of many studies[60] have found that most subject ratings are based on an ordinal scale (i.e., there is no fixed relationship between the difference between a rating of 2 and 3, and a rating of 3 and 4), that some subjects will be overly generous or miserly in their rating, and that without strict rating guidelines different subjects apply different criteria when making their judgements (which can result a subject providing a list of ratings that is inconsistent with all other subjects).

There is no guarantee that data can be fitted, using this model. A study by Wohlin, Runeson and Brantestam[1954] investigated the faults found in an 18-page document by student and professional developers. Your author was unable to fit an IRT model to the data; see Github–regression/stvr95.R.

When software systems are built by a small group of people, the developers may be called on to solve a wide range of computer related issues, including the testing and tuning of the user interface. The System Usability Scale (SUS)[257,258] is a widely used usability questionnaire that produces a single number for usability. One study[1832] compared five methods of evaluating website usability, and found that SUS produced the most consistent results for smaller sample sizes.

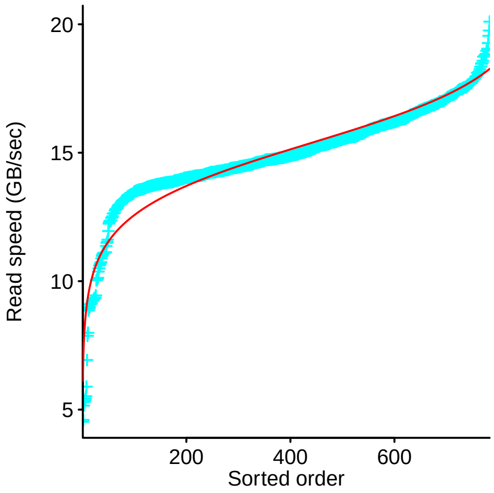


Figure 13.32: Performance of PassMark memory benchmark on 783 Intel Core i7-3770K systems; line is fitted logit model. Data kindly supplied by Wren.[1960] Github–Local

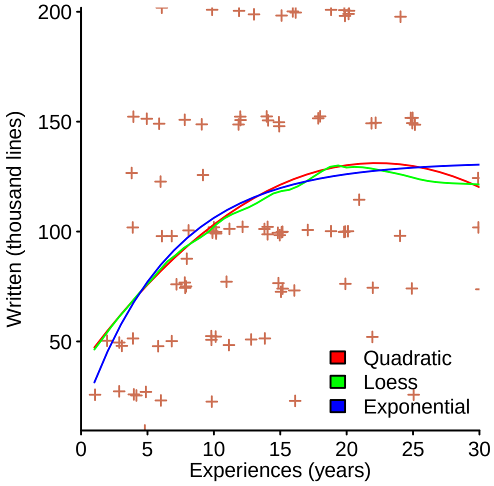


Figure 13.33: Number of lines of code that 101 professional developers, with a given number of years experience, estimate they have written, lines are various regression fits. Data from Jones.[920,921,923] Github–Local

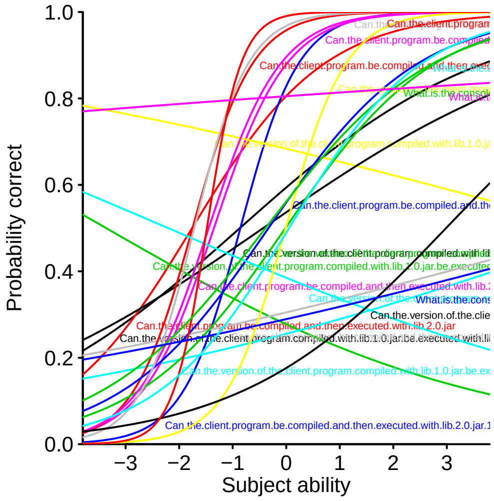


Figure 13.34: Probability that a subject, having a given relative ability, will answer a question correctly: lines are for each question in a fitted Rasch model. Data from Dietrich et al.[490] Github–Local

Chapter 14

Data preparation

14.1 Introduction

The most important question to keep asking yourself while examining, preparing and analyzing any data is: Do I believe this data?

Do patterns appear where none are expected, are expected patterns absent, are human errors missing from the raw data, does the data collector believe whatever they are told, does the measurement process create incentives for people to game it?

Books and presentations on data analysis rarely mention that a large percentage of the time spent on data analysis often has to be invested in data preparation (perhaps 80%, or more, of analysis effort), getting data into a form suitable for the chosen statistical analysis techniques (or statistical package).[i]

Perhaps the largest task within data preparation is data cleaning; an often overlooked[1125] aspect of data analysis that is an essential part of the workflow needed to avoid falling foul of the adage: garbage in, garbage out.

Domain knowledge is essential for data cleaning; patterns have to be understood in the context in which they occur. The fact that many data cleaning activities are generic does not detract from the importance of domain knowledge. For instance, software knowledge tells us that 1.1 is not a sensible measurement value for lines of code (this appears in the NASA MDP dataset), talking to developers at a company to discover they don't work at weekends (e.g., the dates in the 7digital data) and knowing that system support staff used the Unix `pwd` command to check that the system was operational (an analysis of job characteristics for a NASA supercomputer[578] has to first remove 56.8% of all logged jobs, which are uses of `pwd`).

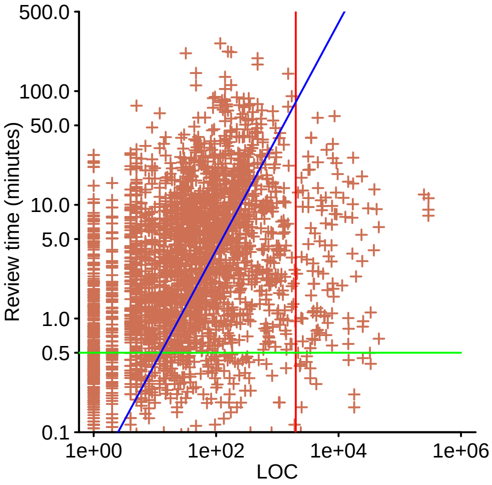


Figure 14.1: Reported LOC and duration of 2,751 code reviews, for one company; reported reviews lasting less than 30 seconds (below green line), involving more than 2,000 LOC (to right of red line), processing at a rate greater than 1,500 LOC per hour (above blue line). Data extracted from Cohen et al.[372] Github–Local

A study by Cohen, Teleki and Brown[372] investigated data from 2,751 code reviews, from one company over a 10-month period. During the data cleaning process they removed all code reviews reported taking less than 30 seconds, involving more than 2,000 LOC and processing code at a rate greater than 1,500 lines per hour. Figure 14.1 shows all reported data points, with points inside the triangle being the only measurements retained for analysis.

Data cleaning is often talked about as-if it is something that happens before data analysis, in practice, the two activities intermingle; the time spent checking and cleaning the data provides insights that lead to a better understanding of the kinds of analysis that might be applicable, also, results from a preliminary analysis can highlight data needing to be cleaned in some way. Data preparation is discussed here in its own chapter, as-if it was performed as a stand-alone activity, in order to simplify the discussion of material in other chapters.

Once cleaned, data may need to be restructured, e.g., rows/columns contained in different files merged into a single data table, or the row/columns in an existing table reorganised in some way. The required structure of the data is driven by the operations that need to be performed on it (e.g., finding the median value of some attribute), or the requirements of the library function used to perform the analysis.

[i] In early versions, this chapter appeared immediately after the Introduction, but in response to customer demand, it was moved here (the penultimate chapter); people want to read about the glamorous stuff, data analysis, not the grunt work of data preparation.

It may be necessary to remove confidential information from the data, or to anonymize information that might be used to identify individuals (or companies). Datasets do not exist in isolation, and it may be possible to combine apparently anonymous datasets to reveal information;[ii] k-anonymity and l-diversity are popular techniques for handling anonymity requirements (in a k-anonymized dataset each record is indistinguishable from at least $k-1$ other records, while l-diversity requires at least l distinct values for each sensitive attribute). Techniques for anonymizing data are not covered here; Fung et al[628] survey techniques for privacy-preserving data publishing, Templ et al[1801] provide an introduction to statistical disclosure control and the `sdcMicro` package.

The behavior of tools used during software development may result in information being lost during some activities. For instance, the Git distributed version control system does not carry-over information from the originator of push/pull requests, and allows commits to be rebased (which changes their history timeline).[662]

While a lot of software engineering data comes from measurements made using software tools, some is still derived from human written records (which can contain a substantial number of small mistakes[911]).

Data cleaning involves a lot of grunt work that often requires making messy trade-offs and having to make do. Tools are available to reduce the amount of manual work involved, but these sometimes require placing trust in the tool doing the right thing; available tools include:

- OpenRefine[1401] (was Google Refine) reads data into a spreadsheet-like form and supports sophisticated search/replace and data transformations editing,
- the `editrules` package checks data values for consistency with user specified rules involving named columns, e.g., `total_fruit ==total_apples+total_oranges`,
- the `deducorrect` package performs automatic value transformations based on user specified consistency rules, relating to column values that must be met (e.g., failure to meet the condition `total_x > 0` will result in any value in that column having a negative sign removed); this package can also impute missing values,
- there are a variety of special purpose packages that handle domain specific data, e.g., the `CopyDetect` package detects copying of exam answers in multi-choice questions.

14.1.1 Documenting cleaning operations

Documenting the changes made to the original data, during the cleaning process, serves a variety of purposes, including:

- enabling third-parties to check that the changes are reasonable and don't produce an unrealistic analysis,
- enabling potential sources of uncertainty to be checked when multiple analysts publish results based on the same dataset, i.e., if there are differences in the results, it is possible to check whether these differences are primarily the result of differences in data cleaning,
- providing confidence to users of the final results of the analysis, that the researcher doing the work is competent, i.e., that cleaning was performed.

Ideally the operations performed on the original data, to transform it into what is considered a clean state, are collected together as a script for ease of replication.

Some cleaning activities are trivial and yet need to be performed to prevent the analysis being overwhelmed by what appears to be many special cases. For instance, an analysis[75] of different company's response to vulnerabilities reported in their products, started from raw data that sometimes contained slightly different ways of naming the same company. The company names data had to be cleaned to ensure that a single name was consistently used to denote each organization (see Github–data-check/patch-behav.R).

The NASA Metrics Data Program (MDP) dataset contains fault data on 13 projects, and has been widely used by researchers (a literature survey for the period 2000 to 2010[762] found that 58 out of 208 fault prediction papers used it). This dataset contains many problems[722] that need to sorted out, e.g., columns with all entries having the same value (suggesting a measurement or conversion error has occurred), duplicate rows, missing

[ii] 63% of the US population can be uniquely identified using only gender, ZIP code and full date of birth.[692]

values (many occurred in rows calculated from other rows and involved a divide by zero), inconsistent values (e.g., number of function calls being greater than the number of operators) and nonsensical values (e.g., lines of code having fractional values).

Despite the non-trivial work needed to clean the MDP dataset, to remove spurious data, the authors of many papers using this dataset, have either not cleaned it, or only given a cursory summary of the cleaning operations[214] (e.g., " . . . removes duplicate tuples . . . along with tuples that have questionable values . . . ", does not specify what values were questionable). Consequently, even although the original dataset is publicly available it is difficult to compare the results published in different papers, because no information is available on what, if any, data cleaning operations were performed; so much of the data is in need of cleaning that any results based on an uncleaned version of this dataset must be treated with suspicion.

See Github–data-check/NASA_MDP-data_check.R for examples of integrity checks performed on the MDP dataset.

It is possible that the values appearing in a sample are correct, but have been misclassified.

A survey of 682,000 unique Android devices in use during 2015, by OpenSignal,[1402] included the screen height and width reported by the device (see figure 14.2, upper plot). Many devices appear to have greater width than height, particularly those with smaller screens. Perhaps the device owners are viewing the OpenSignal website with their phones in landscape mode; the lower plot in Figure 14.2 switches the dimensions so that height has the larger value; switched values in red.

The fault repositories of open source projects are publicly available, and the repositories of larger projects are a frequent source of data for fault analysis/prediction researchers.

A study by Herzig, Just and Zeller[811] manually classified over 7,000 issue reports extracted from the fault repositories of seven large Java projects. They found that on average 42.6% of reports had been misclassified, with 39% of files marked as defective not actually containing any reported fault (any fault prediction models built using the uncleaned data are likely to be misleading at best and possibly very wrong). An earlier analysis[812] had found that between 6 and 15% of bug fixing changes addressed more than one issue.

Possible reasons for the misclassification include: the status of an issue not being specified when the initial report is filed, resulting in the default setting of *Bug* being used; issue submitters having the opinion that a missing feature is a bug (request for enhancement was the most commonly reclassified status); and bug reporting systems only supporting a limited number of different issue statuses (forcing the submitter to use an inappropriate status).

The study also highlighted how much effort data cleaning consumes; the work was performed independently by two people and took a total of 725 hours (90 working days).

Sometimes the measured values from one or more subjects (e.g., people or programs) are remarkably different from the values measured for other subjects. It can be tempting to clean the data by removing the value for these subjects from the sample. A study by Müller and Höfer[1308] removed data on seven out of 18 subjects, because they considered the performance of these subjects was so poor that they constituted a threat to the validity of the experiment (whose purpose was to compare the performance of students and professional developers). This kind of activity might be classified as outlier removal or manipulating data to obtain a desired result, either way, documenting the cleaning activity makes it possible for readers of the analysis to decide.

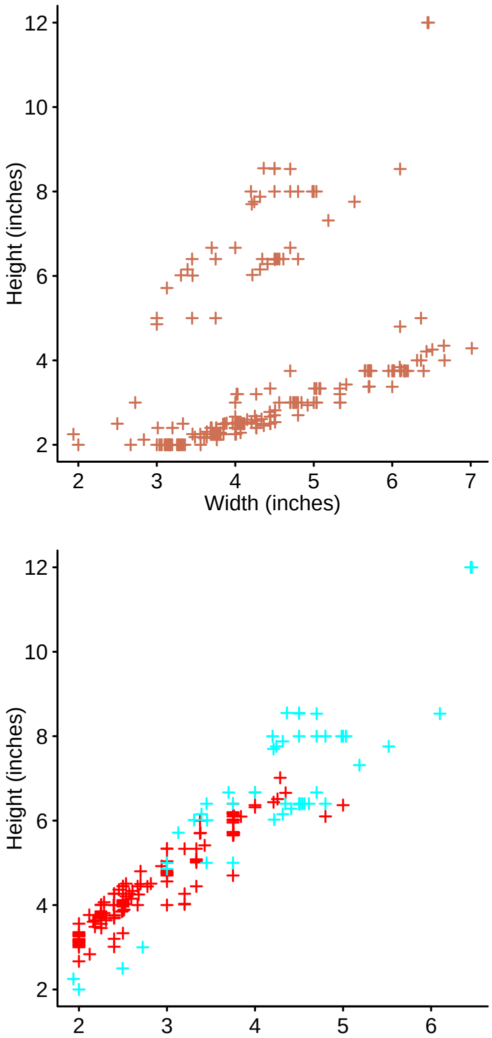


Figure 14.2: Screen height and width reported by 682,000 unique devices that downloaded an App from OpenSignal in 2015 (upper), reported measurements ordered so height always the larger value (lower). Data from OpenSignal.[1402] Github–Local

14.2 Outliers

"An outlier is an observation which deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism".[782]

In some applications, observations that deviate from the general trend are the ones of interest,[313] e.g., intrusion detection and credit card fraud. This subsection covers the case where deviate observations are unwanted; some later subsections cover software engineering situations where deviate observations are themselves the subject of study.

Methods for handling outliers include:

- using a statistical technique that does not assign too much weight to observations that deviate from the patterns followed by most other observations. Techniques capable of performing the desired statistical analysis are not always available, but when R functions implementing them are available they may be discussed in the appropriate section,
- detecting and excluding outliers from the subsequent analysis. Traditionally outliers have been manually selected and excluded from subsequent analysis. This approach can work well when the sample contains a small amount of data, and the person doing the detection has sufficient domain knowledge. There are a variety of functions that automate the process of outlier selection and handling, some of these are discussed below (also see section 11.2.6).

another definition of outlier detection is " . . . the problem of finding patterns in data that do not conform to the expected normal behavior."[313] This definition requires that an expected normal behavior be known, along with a method of comparing values for *outlyingness*.

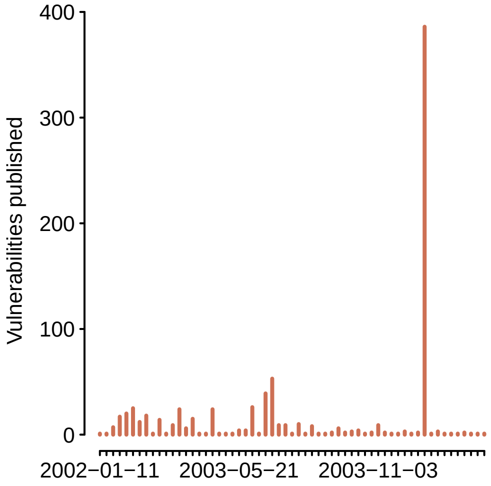


Figure 14.3: Number of reported vulnerabilities, per day, in the US National Vulnerability Database for 2003. Data from the National Vulnerability Database.[1366] Github–Local

Figure 14.3 shows a suspicious spike in the number of daily reported vulnerabilities recorded in the US National Vulnerability Database for 2003. What behavior could explain this pattern? Perhaps all vulnerabilities that had been reported, but not yet fully processed, were simply published, for the public to see, at the end of the year?

A study by Zheng, Mockus and Zhou[1998] investigated what they called problematic values, in the task completion time for Mozilla projects. A manual analysis found that for various reasons, some patches for reported faults were being committed in batches (so the commit date did not reflect the date the code for the patch was created). Enough information was available (i.e., there was data redundancy) to build a model that suggested values more likely to be correct (50% more accurate was claimed).

The date when an event occurred may appear unlikely, based on domain knowledge, e.g., staff rarely work at weekends. The following output shows a count of the number of features recorded as being Done, in a company using an Agile process,[1] for each day of the week. Monday is day 0, and the counts for Saturday/Sunday should be zero; the non-zero values suggest a 2-4% error rate, comparable with human error rates for low stress/non-critical work. Github–Local

```
> table(Done_day %% 7)
  0   1   2   3   4   5   6
670 708 669 716 447  12  16
```

Should outliers be removed from the sample used for analysis?

While removing outliers may improve the quality of the model fitted to an equation, does it improve the quality of the fit of the model to reality?

Without understanding the processes that generated the data, there is no justification for removing any value.

The real issue with outliers is the impact they have on the final result. In a large sample, a few unusual values are unlikely to have any real impact data.

However, outliers are handled, any decision to exclude them from analysis needs to be documented.

14.3 Malformed file contents

A sign that data, or its organization in a file, is malformed in some way, is that the variable into which a file has been read does not have the expected contents (e.g., incorrect number of columns, or a surprising type for the data in one or more columns, e.g., a string where a number was expected). The `str` function provides a quick and easy way of checking the types of columns in a data frame.

File formatting issues to watch out for include:

- functions for reading data in R (e.g., `read.csv` and `read.table`) often use the first few lines of the file being read as the format to use when reading the rest of the file, i.e., the number of columns contained in each row and the datatype of the values in each column. If there are one or more rows that do not follow the format selected at the start of the file (e.g., different number of column delimiters; perhaps the result of non-delimited strings such as a missing pair of quote characters), then subsequent values may appear in the other columns, or be converted to a different type,

- termination delimiter missing from a string value; this can result in the contents of the following row being treated as part of the current row (because the newline is treated as part of the string),
- cut-and-pasting of data between media introducing conversion errors, e.g., the digit zero treated as the letter D or G during image to character conversion.

A variety of ad-hoc techniques are available for locating the cause of problems. For instance, the following code will convert all values that do not have the format of a number to NA, which are then easily located using base-library support for processing NAs, with their row index found using the which function:

```
which(is.na(as.number(as.character(data_frame$column_name))))
```

The complete.cases function returns a vector specifying which rows in its data.frame argument are complete, i.e., do not contain any NAs; the na.omit function returns a copy of its argument with any rows containing NA omitted.

14.4 Missing data

Missing data (for instance, a survey where the entry for a person's age is empty) is often the rule, rather than the exception, and books have been written on the subject. Missing data may be *disguised*, in that it appears as a reasonable value[1440] (e.g., zero when the range of possible legitimate values includes zero), or it may not be visible to the measurement process (e.g., intermittent check-ins to version control obscuring the detailed change history[1342]), or the input process provides a two item choice (e.g., male/female), with one item being the default and thus appearing as the missing value when no explicit choice is made.

The starting point for handling missing data is to normalise how it is denoted, to the representation used by R, i.e., NA (Not Available). Normalisation ensures that all missing values are treated consistently; special case handling of NA is built into R and many functions include options for handling NA.

A wide variety of different representations for missingness may be encountered (e.g., special values that cannot occur as legitimate data values, such as: 9999, "#N/A", "missing", or no value appearing between two commas in a comma separated list), and it is not uncommon for different columns within a dataset to use different representations (because they originate from different measurement sources).

The following code illustrates one method for changing a known representation of missing value to NA (the second form would be necessary if 9999 could appear as a legitimate value in a column other than size):

```
data[ data == 9999 ] = NA  # set all elements having value 999 to NA

# set all elements of column size having value 999 to NA
data$size[ data$size == 9999 ] = NA
```

Once missing values have been explicitly identified it is possible to move on to deciding whether to ignore these cases or to replace NA with some numeric value. Some algorithms can handle missing values while others cannot; R functions vary in their ability to handle missing values. A few techniques for selecting the replacement value are discussed below.

The R base I/O functions, such as read.csv, have conventions for handling the case of zero characters appearing between the delimiters on each line of a file. The behavior depends on the type that has been assigned to values in a particular column. For columns assigned a numeric type, zero characters are treated as-if NA appeared between the delimiters, while for columns assigned a string type the zero character case is treated as the empty string rather than NA (i.e., treated the same as the string ""). These functions support a variety of options for changing the default the handling of zero characters and the handling of leading/trailing white-space between delimiters.

As an example: reading a file containing the columns below left has the same effect as reading a file containing the columns below right:

```
X,Y_str,Z             X,Y_str,Z
1,"abc",2.2           1,"abc",2.2
2,,3.1                2,"",3.1
,NA,2                 NA,NA,2
```

The table function counts occurrences of values, and by default does not include NA in the count; the useNA options has to be used to explicitly specify that NA be counted:

```
table(data$some_column, useNA="ifany") # limit the count to one column
```

This one column use can be expanded to cover every column in data. If the output is too voluminous, the number of columns processed can be reduced, or the call to table replaced by a call to tabulate, which provides more options to control behavior:

```
sapply(colnames(data), function(x) table(data[ , x], useNA="ifany"))
```

While there may be documentation specifying how missing values are represented, such details may not be documented. An analysis of a dataset using the above code may show a suspiciously large number of values such as 9999 or -1 (for an attribute that can never be negative), a result that suggests further investigation is worthwhile.

14.4.1 Handling missing values

When deciding what to do about missing values, it is important to try to understand why the values are missing. The following categories are commonly encountered in the analysis of missing data:

- Missing completely at random (MCAR): As the name suggests, the selection of missing values occurred completely at random. Statistically this is the most desirable kind of missingness, because it means there is no bias in the missing values,
- Missing at random (MAR): This sounds exactly like MCAR, but it is not completely random in the sense that the choice of which values are missing is influenced by other values in the sample. For instance, the level of seniority may correlate with the likelihood that survey questions about salary are answered,
- Missing not at random (MNAR): This missingness could be as random as MAR, with the one difference that the choice of missing values is influenced by values not in the sample. For instance, the name of the developer who originally wrote the code referenced in a fault report may be missing if that developer is friendly with the person reporting the fault, with friendship not being a recorded in the sample.

The following code can be used to get a rough estimate of the correlation between the rows of a data.frame that contain missing values (figure 8.8 illustrates a method of visualizing this information):

```
x=is.na(some_data_frame)
# highlight rows having some, but not all, missing values
cor(subset(x, sd(x) > 0))
```

Many analysis techniques handle missing values by ignoring the rows or columns that contain them; if the sample contains many rows and a low percentage of missing values, this behavior may not be a problem. However, if the sample contains a large percentage of missing values, any analysis will either have to make do with a smaller number of measurements, be limited to using techniques that can gracefully adapt to missing data (i.e., don't ignore rows containing one or more missing values) or be forced to use estimated values for the missing data.

The ideal approach is to use an algorithm capable of handling samples that include missing data.

The process of estimating a value to use, where none is present in the sample, is known as *imputing*.

A quick and dirty method of imputing values, that can be effective, is to replace a missing value by the mean of the values in the corresponding column containing the missing value; alternatively, if the data is ordered in some way (e.g., dates), the last value appearing before the missing value might be used.

A more sophisticated approach to imputing values involves filling the missing value entries using other values present in the sample. The Amelia, naniar and VIM packages provide a variety of functions for visualizing datasets containing missing values and imputing values for these entries.

A study by Buettner[269] investigate project staffing, but was not able to obtain complete staffing information. Figure 14.4 shows a loess fit and the 95% confidence bounds.

Some R functions support the use of splines for interpolating values. Splines originated as a method for connecting a sequence of points by a smooth curve, not as a method for fitting a curve minimizing some error metric. Apart from their familiarity, there is no reason to prefer the use of splines over other techniques (implementation issues also exist with the `bs` and `ns` functions, in the `splines` package, when fitting a model with the `predict.glm` function[1869] and then making predictions using new data points).

Data may be missing because the sample may not be large enough to be likely to contain instances of rarely occurring cases (which would be seen in a larger sample). Good-Turing smoothing[635] is a technique for adding non-zero counts to adjust for unseen items.

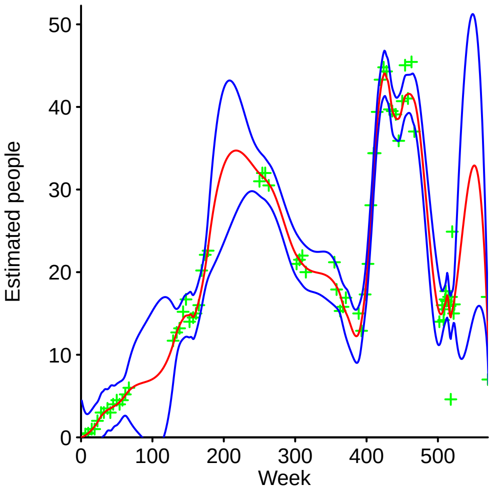


Figure 14.4: Estimated staff working on a project during each week; lines are a fitted loess model and 95% confidence bounds. Data from Buettner.[269] Github–Local

14.4.2 NA handling by library functions

R functions vary in their ability to handle `data.frames` containing `NA`, with the behaviors exhibited including:

- behaving in unpredictable ways when `NA` is encountered,
- behaving in predictable ways, that perhaps is surprising to the unknowledgeable, e.g., the value of `NA ==NA` is `NA`, as is `NA !=NA`,

  functions that operate on complete rows or columns have a variety of behaviors when they counter one or more `NA`s, including:

  - supporting a parameter, often called `na.rm`, which can be used to select among various methods for handling any `NA` that occur,
  - ignoring rows containing one or more `NA`, e.g., `glm` ignores these rows by default, but this behavior can be changed using the `na.action` option,
  - making use of information present in rows containing one or more `NA`, e.g., the `rpart` function,

Some regression model building functions return information associated with individual data points, such as residuals. If the function removes rows containing any `NA` before building the regression model, the number of data rows included in the returned model may be less than originally passed in, unless rows containing `NA` is reinserted (e.g., by using the `naresid` function).

14.5 Restructuring data

When the data of interest is spread over several files, it may be necessary to read two or more files and merge their contents into a single `data.frame`.

If two datasets contain shared columns (i.e., column names, column ordering and information held are the same), the `rbind` function can be used to join rows together, returning a single `data.frame`; the `cbind` function performs the join operation for columns.

The `merge` function merges the contents of two `data.frames` based on one or more criteria, e.g., shared column names.

14.5.1 Reorganizing rows/columns

The organization of rows and columns in a `data.frame` may not be appropriate for that used by the library functions used to perform the analysis.

The values in a dataset are may be held in a wide format (i.e., a few rows and many columns), but a long format (i.e., many rows and a few columns) is required, or vice versa.

An example of wide format data is that used in figure 2.5; the IQ test scores have the form:

```
test,gender,1,2,3,4,5,6,7,8,9
verbal,Boy,8455,14171,17596,29308,30490,27544,16037,9857,4635
verbal,Girl,5448,10570,15312,28591,32385,30830,18557,11443,5321
quantitative,Boy,3138,19634,18258,29037,23255,30376,16504,12565,5095
quantitative,Girl,2313,16905,19002,32707,26438,32413,15215,10007,3406
non-verbal,Boy,1390,18144,20713,29245,25720,27077,18095,11369,6077
non-verbal,Girl,1165,14370,18564,30488,29342,30458,18387,10450,5075
CAT3,Boy,2505,14505,19556,29917,29607,30327,17960,9392,2787
CAT3,Girl,1813,10927,17872,31059,32867,33269,18016,9041,2394
```

The `melt` function, in the `reshape2` package, transforms `data.frames` to a long format, such as the following (only the first 11 lines are shown):

```
           test gender stanine count
1        verbal    Boy      X1  8455
2        verbal   Girl      X1  5448
3  quantitative    Boy      X1  3138
4  quantitative   Girl      X1  2313
5    non-verbal    Boy      X1  1390
6    non-verbal   Girl      X1  1165
7          CAT3    Boy      X1  2505
8          CAT3   Girl      X1  1813
9        verbal    Boy      X2 14171
10       verbal   Girl      X2 10570
11 quantitative    Boy      X2 19634
```

which was reorganized using the call (where `b_g_IQ` contains the data):

```
b_g=melt(b_g_IQ, id.vars=c("test", "gender"),
                 variable.name="stanine", value.name="count")
```

It is also possible to convert from long to wide format.

14.6 Miscellaneous issues

14.6.1 Application specific cleaning

The analysis of some kinds of data has acquired established preprocessing procedures; the data is not wrong, but transforming it in some way improves the quality of subsequent analysis. For instance, before analyzing text, common low interest words (such as “the” and “of”, known as *stop words*) are removed; also words may be stemmed (a process that removes suffixes with the intent of uncovering the root word, e.g., kicked and kicking both become kick).

14.6.2 Different name, same meaning

Typos in character based data may be detected because of constraints on what can appear in domain specific sequences (e.g., the spelling of words). More difficult to detect problems include different people using different terminology for the same concept, or the same terminology for different concepts.

The SPEC 2006 benchmark results often include a description of the characteristics of the memory used by the computer under test. For historical marketing reasons, two scales are commonly used to specify memory performance; the DDR scale is based on peak bandwidth, while the PC scale uses clock rate. The SPEC result descriptions are not consistent in their choice of scale, and so before any analysis can be performed the values have to be converted to a single form. Also, for marketing reasons, the values are rounded to reduce the number of non-zero digits; an analyst interested in high accuracy would map the *marketing* values to their actual values (see Github–benchmark/scripts/SPEC-memory.awk).

An email address is sometimes the only unique identifying information available, e.g., the list of developers who have contributed to an open source project. The same person may have used more than one email address over the period of their involvement in a project, and it is necessary to detect which addresses belong to the same person.[1936]

14.6.3 Multiple sources of signals

Sometimes a value appearing in a sample could have come from multiple sources, only one of which is of interest. An example of this is the question: when did hexadecimal literals first appear as such in print?

One way of answering this question is to analyze the word n-grams (and associated year of book publication) Google have made available from their English book scanning project.[700]

The regular expression `^[0oO[xX][0-9a-fA-FoOl]]` (ohh, Ohh and ell were treated as the corresponding digits) returned 89 thousand matches.

OCR mistakes have resulted in some words being treated as hexadecimal literals, e.g., Oxford was sometimes scanned as 0xf0fd. The character sequence oxo is common, and looking at some of the contexts in which this sequence occurs suggests that the usage is mainly related to chemical formula (some uses are also likely to be references to a cooking product of this name).

Assuming that hexadecimal notation did not start appearing in books before electronic computers were invented, books prior to say 1945 (i.e., the end of World War II) can be ignored.

Your author also assumed that, if any hexadecimal literal appears in a book, at least one more such literal is likely to appear; applying this final filtering rule, the number of matches was reduced to 7,292; with 319 unique character sequences.

Figure 14.5 shows a comparison of the use of hexadecimal literals in C source with those extracted from Google books n-grams.

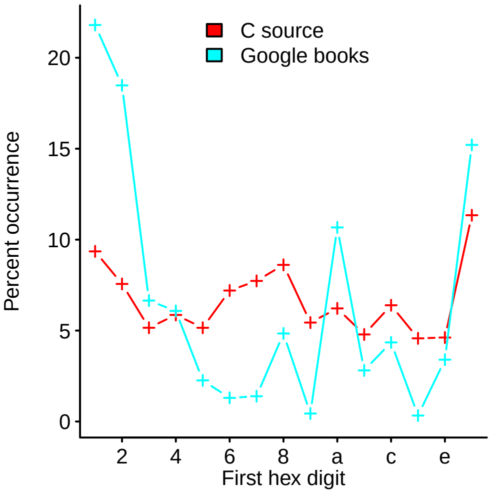


Figure 14.5: Percentage occurrence of the first digit of hexadecimal numbers in C source and estimated from Google book data. Data from Jones[919] and Michel et al.[1259] Github–Local

14.6.4 Duplicate data

Duplicate data can cause some analysis techniques to fail (e.g., regression modeling) or skew the calculated results.

Duplication data is easily generated: the collation of data from multiple sources can result in the same measurements appearing more than once, and there may be multiple measurements of the same event (e.g., logging of computer faults where a single root cause produces the same message at sporadic times after the fault is experienced[1786] and spatially or functionally adjacent units to generate messages;[1120] see Github–data-check/Blue-Gene.log).

The `duplicated` function returns information about rows that are exact duplicates. More subtle duplication may involve the values in a row/column differing by a constant factor from those in another row/column, e.g., one row contains temperature in Celsius while another uses Fahrenheit.

When the data is numeric, *close* duplicates can be highlighted using pairwise correlation; see section 10.5.4.

Some R functions handle duplicate row/columns gracefully (e.g., the `glm` function), while others give unpredictable results (e.g., the `solve` function, which inverts a matrix), the behavior depends on the algorithm used and what if any consistency checks were added by the implementer of the code.

14.6.5 Default values

Sometimes a measurement process returns what is considered to be a reasonable value, if it cannot return the actual value. For instance, IP geolocation services are always able to associate a country with an IP address, but when they are unable to further refine the location within a country, they return a location near the center of the country; for the USA this is close to the town of Potwin in Kansas (population 449) which appears to experience orders of magnitude more Internet related events, for its population size, than other towns in the US.[897]

14.6.6 Resolution limit of measurements

Some kinds of measurement are inherently inexact, e.g., time. When working close to the resolution limit of the measuring process, false signals can be generated by the interaction between the measurement resolution and the processes generating measurement events.

A study by Feitelson[577] measured the runtime of processes, executed on a system, to an accuracy of two decimal digits. Initial analysis of the number of processes whose execution fell within a given time interval found an unexpected behavior, there were many time intervals that did not contain any processes (see Figure 14.6, upper plot). Further analysis found that the timer resolution was $1/64$ second, and the gaps were an artefact of the number of digits recorded, recording more digits (see Figure 14.6, lower plot) resulted in fewer intervals containing no measurement points.

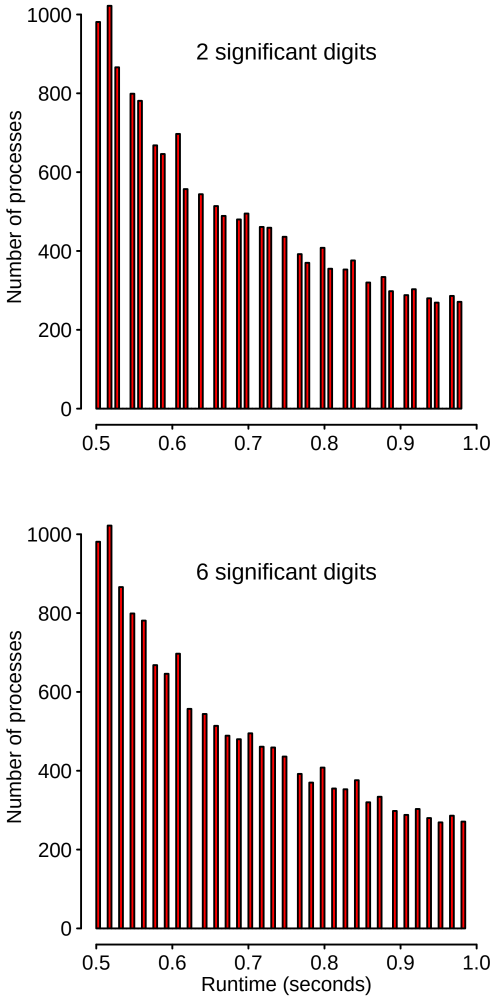


Figure 14.6: Number of processes executing for a given amount of time, with measurements expressed using two and six significant digits. Data from Feitelson.[577] Github–Local

14.7 Detecting fabricated data

A sample is not always derived from accurate measurements, the accuracy failures may be accidental or intentional, or might not involve any actual measurements (i.e., it has been fabricated).

Like all data analysis, detection of fabricated data is based on finding known patterns in the data (i.e., patterns that have previously been found to appear in known fabricated data). As always, the interpretation of why the data contains these patterns is the responsibility of the audience of the results; it is always worth repeating that domain knowledge is key.

One pattern of behavior observed in real world data, with some regularity, is the first digit of numeric values following Benford's law to a reasonable degree of approximation (while a figure of 30% of all datasets has been quoted, the actual figure is likely to be much smaller[1641]). The failure of data to follow Benford's law has been used to detect accounting and election[1588] fraud, identification of fake survey interviews[1632] and scientific research.[489]

While references to Benford's law usually involve the first digit of numeric values, there is a form that applies to the second and perhaps other significant digits.[1362] There has also been work[153] suggesting that the digit at the opposite end of numeric literals, the least significant digit, sometimes has a uniform distribution.

Benford's law specifies that the probability of the first digit having value d is given by: $P(d) = \log_{10}(1 + \frac{1}{d})$

Figure 7.51 shows percentage occurrences for the first digit of numeric literals in C source code.

If a set of independent and identically distributed random variables are sorted, the distribution of digits of the differences between adjacent sorted values is close to Benford's law.[1270] A test based on this fact can detect rounded data, data generated by linear regression and data generated by using the inverse function of a known distribution.[1362]

The `BenfordTests` package contains a variety of function for evaluating the conformity of a dataset to Benford's law.

When generating fabricated data, it is sometimes necessary to produce a random sequence of items. People hold incorrect beliefs[208] about the properties of random sequences and when asked to generate them produce sequences that contain predictable patterns (i.e., they are not random).

One study[1635] was able to build a model that predicted repeated patterns in an individual's *randomly* selected sequence, with around 25% success rate, but when the model built for one person's behavior was used to make predictions about another persons the success rate dropped to around 18%.

Detecting divergence from, or agreement with, these patterns of behavior depends on the authors of the data being unfamiliar with the expected patterns of behavior, or being lazy (i.e., being unwilling to spend the time making sure that the data they generate has the expected characteristics; the creators of the fictitious accounts publicly published by Madoff's companies, before his fraud was uncovered, made the effort to ensure they followed Benford's law[1646]).

An excess of round numbers has been used to suggest that data has been fabricated.[1019]

If people are willing to invest some effort, it is possible to manipulate data such that some statistical tests meet expectations;[1204] see Github–communicating/warp-pts.R.

Chapter 15

Overview of R

This chapter gives a brief overview of R for developers who are fluent in at least one other computer language. The discussion pays attention to language features that are very different from languages the reader is likely to be familiar with; the focus is on a few language features that can be used to solve most problems.

The R language is defined by its one implementation; available from the R core team.[1527] A language definition,[1526] written in English prose, is gradually being written.

R programs tend to be very short, compared to programs in languages such as C++ and Java; 100 lines is a long R program. It is assumed that most readers will be casual users of R, whose programs generally follow the pattern:

```
d=read_data()
clean_d=clean_and_format(d)
d_result=applicable_statistical_routine(clean_d)
display_results(d_result)
```

If your problem cannot be solved using this algorithm, then the most efficient solution may be for you, dear reader, to use the languages and tools you are already familiar with, to preprocess the data so that it can be analysed and processed using R.

R is a domain specific language, whose designers have done an excellent job of creating a tool suited to the tasks frequently performed when analysing the kinds of datasets encountered in statistical analysis. Yes, R is Turing complete, so any algorithm that can be implemented in other programming languages can be implemented in R, but it has been designed to do certain things very well, with no regard to making it suitable for general programming tasks.

As a language the syntax and semantics of R is a lot smaller than many other languages. However, it has a very large base library, containing over 1,000 functions. Most of the investment needed to become a proficient user of R has to be targeted at learning to how to combine these functions to solve the problem at hand. There are over 10,000+ add-on packages available from the CRAN (Comprehensive R Archive Network).

Help on a specific identifier, if any is available, can be obtained using the `?` (question mark) unary operator, followed by the identifier. The `??` unary operator, followed by the identifier, returns a list of names associated with that identifier for which a help page is available.

The call `library(help=circular)`, lists the functions and objects provided by the package named in the argument.

15.1 Your first R program

Much like Python, Perl and many other interpreted implementations, R can be run in an interactive mode, where code can be typed and immediately executed (with `"Hello world"` producing the obvious output).

Your first R program ought to read some data and plot it, not just print `"Hello World"`. The following program reads a file containing a single column of values and plots them, to produce figure 15.1:

```
the_data=read.csv("hello_world.csv")

plot(the_data)
```

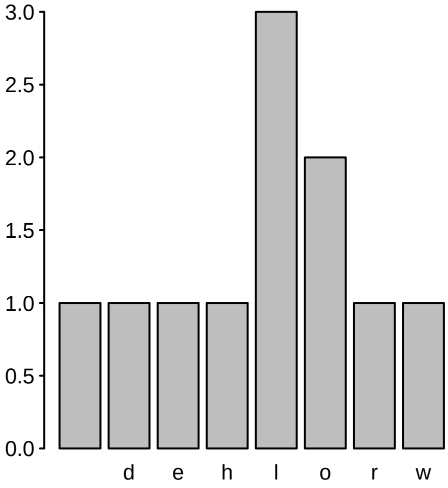


Figure 15.1: Plot produced by hello_world.R program. Github–Local

The `read.csv` function is included in the library that comes bundled with the base system (functions not included in this library have to be loaded using the `library` function, before they can be referenced; the package containing them may also need to be installed via the `install.packages` function) and has a variety of optional arguments (arguments can be omitted if the function definition includes default values for the corresponding parameter). Perhaps the most commonly used optional arguments are `sep` (the character used to separate, or delimit, values on a line, defaults to comma) and `header` (whether the contents of the first line should be treated as column names, default `TRUE`).

The value returned by `read.csv` has class `data.frame`, which might be thought of as a C `struct` type (it contains data only, there are no member functions as such).

The `plot` function attempts to produce a reasonable looking graphic of whatever data is passed, which for character data is a histogram of the number of occurrences. Users of R are not expected to be interested in manipulating low level details, and some effort is needed to get at the numeric values of characters.

There are a wide variety of options to change the appearance of `plot` output; these can be applied on each call to `plot`, or globally for every call (using the `par` function).

All objects in the current environment can be saved to a file using the `save` function, and a previously saved environment can be restored using the `load` function. When quiting an R session (by calling `q()`), the user is given the option of saving the current environment to a file named `.RData`; if a file of this name exists in the home directory, when R is started, its contents are automatically loaded.

15.2 Language overview

R is a language and an environment. Like Perl, it is defined by how its single implementation behaves (i.e., the software maintained by the R project[1527]).

R was designed, in the mid-1990s, to be largely compatible with S (a language, which like C, started life in the mid-1970s at Bell Labs). When S was created, Fortran was the dominant engineering language and the Fortran way of doing things had a strong impact on early design decisions (i.e., R does not have a C view of the world; for instance, it uses a row/column, rather than column/row ordering).

The designers of R have called it a functional language, and it does support a way of doing things that is most strongly associated with functional program languages (including making life cumbersome for developers wanting to assign to global variables).

The language also contains constructs that are said to make it an object-oriented language, and it certainly contains some features found in object-oriented languages. Object-oriented constructs were first added in the third iteration of the S language, and were more of an addition to the functional flavor of the language than a complete make-over. The primary OO feature usage is function overloading, when accessing functions from library packages.

Lateral thinking is often required to code a problem in R, using knowledge of functions contained in the base system, e.g., calling `order` to map a vector of strings to a unique vector of numbers.

Some data analysts write non-trivial programs in R, which means they have to deal with the testing and debugging issues experienced by users of other languages.

Base library support for debugging includes support for single stepping through a function, via the `debug` function, and setting breakpoints via the `setBreakpoint` function; package support includes: `RUnit` package for unit testing, and the `covr` package for measuring code coverage.

15.2.1 Differences between R and widely used languages

The following list describes language behaviors that are different from that encountered in other commonly used languages (Fortran developers will not consider some of these to be differences):

- there are no scalars, e.g., 2 is a vector containing one element and is equivalent to writing c(2). Most unary and binary operators operate on all elements of a vector (among other things).

  Many operations that involve iterating over scalar values in other languages, e.g., adding two arrays, can be performed without explicit iteration in R, e.g., c(1, 2) + c(3, 4) has the value c(4, 6),
- arrays start at one, not zero,
- matrices and data frames are indexed in row-column order (C-like languages use column-row order),
- case is significant in identifiers, e.g., some_data and Some_data are considered to refer to different objects,
- the period (dot, full stop, i.e., .) is a character than can occur in identifiers (e.g., a.name), it is not a separate token having the role of an operator,
- some language constructs, implemented via specific language syntax in other languages, are implemented as function calls in R, e.g., the functionality of return and switch is provided by function calls,
- assignment to a variable in an outer scope, from within a function, is specified using the **<<-** operator. The other assignment operators (e.g., **<-**, **->** and =) always assign to a local variable (creating one, if a variable of the given name does not already exist, in local scope),
- vectors/arrays/data.frames can be sliced to return a subset of the original,
- explicit support for NA (Not Available). This value denotes a number that may exist, but whose value is unknown. Operations involving NA, return NA, when the result value is not known because the value of NA is unknown, but will return a value when the result is independent of the value of NA, e.g., NA || TRUE,
- type conversion behavior may be driven by semantics rather than the underlying representation, e.g., as.numeric("1") ==1 and as.numeric("a") returns NA.

The following R language features are found in commonly used languages:

- objects and functions come into existence, during program execution, when they are assigned a value, appear as a function parameter, or in more obscure ways (there is no mechanism for declaring any kind of identifier),
- the type of an object is the type of the value last assigned to it,
- decimal and hexadecimal literals have type numeric (literals starting with zero are not treated as octal literals; any leading zero is ignored) even if they look like integers, because they do not contain a decimal point. Some input functions, e.g., read.csv, consider a column to have integer type, if all its values can be represented as an integer,
- what most other languages consider to be a statement (i.e., something that does not return a value) R treats as an expression (e.g., if/for statements return a value).

15.2.2 Objects

Operations in R are performed on objects, sometimes known as variables. Objects are characterized by their names and their contents; with the contents in turn being characterized by attributes specifying the kind of data contained in the object.

The R type system has evolved over time, and includes the terms *mode* (a higher level view of the value representation, at least sometimes, than *typeof*, e.g., integer and double have mode numeric), *storage.mode* (a concept going back to the S language) and *typeof* (the underlying representation used by the C implementation of the language).

The mode, storage.mode and typeof functions return a string containing the respective information, e.g., numeric, integer or function.

The length of an object is the number of elements it contains (e.g., a two-dimensional array containing i rows and j columns, contains $i \times j$ elements). The length function returns the number of elements in its argument.

The assignment operator creates an object, with the object name being the left operand and its value and type being that of the right operand (left/right is reversed when the **->** assignment operator is used).

15.3 Operations on vectors

15.3.1 Creating a vector/array/matrix

An R vector can be thought of as a one-dimensional array. Vectors are indexed starting at 1 (not zero), and it is possible to added additional elements to a vector, but not remove an existing element.

```
x = 2                   # new vector containing one value
x = c(2, 4, 6, 8, 10)   # new vector containing five values
# new vector containing the contents of x and two values
x = c(x, 12, 14)
y = vector(length=5)    # new vector created by function call
y = 3:8                 # same as c(3, 4, 5, 6, 7, 8)
z = seq(from=3, to=13, by=3)      # create a sequence of values
# All elements converted to a common type
z = c(1, 2, "3")        # String has the greater conversion precedence
```

Multidimensional arrays can be created using the `array` function, with the common case of 2-dimensional arrays supported by a specific function, i.e., the `matrix` function.

```
> # create 3-dimensional array of 2 by 4 by 6, initialized to 0
> a3=array(0, c(2, 4, 6))
> matrix(c(1, 2, 3, 4, 5, 6), ncol=2) # default, populate in column order
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> # specify the number of rows and populate by row order
> matrix(c(1, 2, 3, 4, 5, 6), nrow=2, byrow=TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> x = matrix(nrow=2, ncol=4) # create a new matrix
> y = c(1, 2, 3)
> z = as.matrix(y)  # convert a vector to a matrix
> str(y)
 num [1:3] 1 2 3
> str(z)
 num [1:3, 1] 1 2 3
```

15.3.2 Indexing

One or more elements of a vector/array/matrix can be accessed using indexing. Accesses to elements that do not exist return NA. Negative index values specify elements that are excluded from the returned value.

The zeroth element returns an empty vector.

```
> x = 10:19
> x[2]
[1] 11
> x[-1]                           # exclude element 1
[1] 11 12 13 14 15 16 17 18 19
> x[12]                           # there is no 12'th element
[1] NA
> x[12]=100                       # there is now
> x
 [1]  10  11  12  13  14  15  16  17  18  19  NA 100
```

Multiple elements can be returned by an indexing operation:

```
> x = 20:29
> x[c(2,5)]                       # elements 2 and 5
[1] 21 24
> y = x[x > 25]                   # all elements greater than 25
> y
[1] 26 27 28 29
```

```
> # The expression x > 25 returns a vector of boolean values
> i = x > 25
> i
 [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
> # an element of x is returned if the corresponding index is TRUE
> x[i]
[1] 26 27 28 29
```

Matrix indexing differs from vector indexing in that out-of-bounds accesses generate an error.

```
> x = matrix(c(1, 2, 3, 4, 5, 6), ncol=2)
> x[2, 1]
[1] 2
> # x[2, 3] need to be able to handle out-of-bounds subscripts in Sweave...
> x[, 1]
[1] 1 2 3
> x[1, ]
[1] 1 4
> x[3, ]=c(0, 9)
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    0    9
> x=cbind(x, c(10, 11, 12))  # add a new column
> x
     [,1] [,2] [,3]
[1,]    1    4   10
[2,]    2    5   11
[3,]    0    9   12
> x=rbind(x, c(5, 10, 20))   # add a new row
> x
     [,1] [,2] [,3]
[1,]    1    4   10
[2,]    2    5   11
[3,]    0    9   12
[4,]    5   10   20
```

15.3.3 Lists

The difference between a list and a vector is that different elements in a list can have different modes (types) and existing elements can be removed.

```
> x = list(name="Bill", age=25, developer=TRUE)
> x
$name
[1] "Bill"

$age
[1] 25

$developer
[1] TRUE
> x$name
[1] "Bill"
> x[[2]]
[1] 25
> x = list("Bill", 25, TRUE)
> x
[[1]]
[1] "Bill"

[[2]]
[1] 25

[[3]]
[1] TRUE
> y = unlist(x) # convert x to a vector, all elements are converted to strings
> y
[1] "Bill" "25"   "TRUE"
> x = list(name="Bill", age=25, developer=TRUE)
> x$sex="M"  # add a new element
> x
$name
[1] "Bill"

$age
[1] 25

$developer
[1] TRUE

$sex
[1] "M"
> x$age = NULL # remove an existing element
> x
$name
[1] "Bill"

$developer
[1] TRUE

$sex
[1] "M"
```

The **[[ ]]** operator returns a value, while **[ ]** returns a sublist (which has mode list).

15.3.4 Data frames

From the perspective of a programmer coming from another language, it may seem appropriate to think of a data frame as behaving like a matrix, and in some cases it can be treated in this way (e.g., when all columns have the same type, functions expecting a matrix argument may work). However, a better analogy is to think of it as an indexable structure type (where different members can have different types).

The `read.csv` functions reads a file containing columns, of potentially different types, and returns a data frame.

When indexing a data frame like a matrix, elements are accessed in row-column order (not the column-row order used in C-like languages). The following code selects all rows for which the num column is greater than 2.

```
> x = data.frame(num=c(1, 2, 3, 4), name=c("a", "b", "c", "d"))
> x
  num name
1   1    a
2   2    b
3   3    c
4   4    d
> # Middle two values of the column named num
> x$num[2:3]
[1] 2 3
> # Have to remember that rows are indexed first and also specify x twice
> x[x$num > 2, ]
  num name
3   3    c
4   4    d
> # Using the subset function removes the possibility of making common typo mistakes
> # No need to remember row/column order and only specify x once
> subset(x, num > 2)
  num name
3   3    c
4   4    d
```

If one or more columns contain character mode values (i.e., strings), `read.csv` will create, by default, a factor rather than a vector. The argument: `as.is=TRUE` causes strings to be represented as such.

Where ever possible, the code written for this book uses the `subset` function, rather than relying on correctly indexing a data frame.

15.3.5 Symbolic forms

An R expression can have a value which is its symbolic form.

```
exp = expression(x/(y+z))
eval(expr) # evaluate expression using the current values of x, y and z
```

Uses of expression values include: specifying which vectors in a table to plot in a graph, and including equations in graphs, for instance:

```
text(x, y, expression(p == over(1, 1+e^(alpha*x+beta))))
```

results in the following equation being displayed at the point (x, y): $p = \frac{1}{1+e^{\alpha x+\beta}}$

The `D` function takes an expression as its first argument, and based on the second argument, returns its derivative:

```
> D(expression(x/(y + z)^2), "z")
-(x * (2 * (y + z))/((y + z)^2)^2)
```

15.3.6 Factors and levels

When manipulating non-numeric values (e.g., names) statisticians sometimes find it convenient to map these values to integer values and manipulate them as integers. In programming terminology, a variable used to represent one or more of these integer values could be said to have a *factor* type, with the actual numeric values known as *levels* (a parallel can be drawn with the enumeration types found in C++ and C, except these assign names to integer values).

```
> factor(c("win", "win", "lose", "win", "lose", "lose"))
[1] win  win  lose win  lose lose
Levels: lose win
```

Some operations implicitly convert a sequence of values to a factor. For instance, `read.csv` will, by default, convert any column of string values to a factor; this conversion is a simplistic form of hashing, and (when a megabyte was considered a lot of memory) was once driven by the rationale of saving storage. These days the R implementation uses more sophisticated hashing, and we live with the consequences of historical baggage.

Operations of objects holding values represented as factors sometimes have surprising effects, for those unaware of how things used to be.

15.4 Operators

Operators in R have the same precedence rules as Fortran, which in some cases differ from the C precedence rules (which most commonly used languages now mimic). An example of this difference is: `!x ==y` which is equivalent to `!(x ==y)` in R, but in C-like languages is equivalent to `(!x) ==y` (if `x` and `y` have type boolean, there is no effective difference, but expressions such as: `!1 ==2` produce a different result).

A list of operators and their precedence can be obtained by typing `?Syntax`, at the R command line.

Within an expression operand evaluation is left to right, except assignment which evaluates the right operand and then the left.

In most cases, all elements of a vector are operated on by operators:

```
> c(5, 6) + 1
[1] 6 7
> c(1, 2) + c(3, 4)
[1] 4 6
```

```
> c(7, 8, 9, 10) + c(11, 12)
[1] 18 20 20 22
> c(0, 1) < c(1, 0)
[1]  TRUE FALSE
```

in the last two examples *recycling* occurs, that is the elements of the shorter vector are reused until all the elements of the longer vector have been operated on.

The **&&** and || operators differ from **&** and | in that they operate on just the first element of their operands, returning a vector containing one element, e.g., `c(0,1) && c(1,1)` returns the vector `FALSE`.

The base system includes a set of `bitw???` functions, that perform bitwise operations on their integer arguments; there is the `bitops` package.

Operators	**Description**
:: :::	access variables (right operand) in a name space (left operand)
$ @	component / slot extraction (member selection has lower precedence than subscripting in C-influenced languages)
[[[	array and list indexing
^ **	exponentiation (associates right to left)
- +	unary minus and plus
:	sequence operator
%any%	special operators (%% and %/% has the same precedence as * and / in C-influenced languages)
* /	multiply and divide
+ -	(binary) add and subtract
< > <= >= == !=	relational and equality (non-associative; equality has lower precedence in C-influenced languages)
!	negation (greater precedence than any binary operator in C-influenced languages)
& &&	and of all elements and the first element
| ||	or of all elements and the first element
~	as in formulae
-> ->>	local and global rightwards assignment
=	assignment (associates right to left)
<- <<-	local and global assignment (associates right to left)
?	help (unary and binary)

Table 15.1: R operators listed in precedence order.

The character used for exclusive-or in C-influenced languages, ^, is used for exponentiation in R; the `xor` function performs an exclusive-or of its operands. Like Fortran, R also supports the use of ** to denote exponentiation.

The `[` and `[[` operators differ by more than being array and list indexing. The result of the index `x[1]` has the same type as `x` (i.e., the operation preserves the type), while the result of `x[[1]]` is a simplified version of the type of `x` (if simplification is possible).[i]

```
> x = c(a = 1, b = 2)
> x[1]
a
1
> x[[1]]
[1] 1
> x = list(a = 1, b = 2)
> str(x[1])
List of 1
 $ a: num 1
> str(x[[1]])
 num 1
> x = matrix(1:4, nrow = 2)
> x[1, ]
[1] 1 3
> x[1, , drop = FALSE]
     [,1] [,2]
```

[i] Out-of-bounds handling is also different, but I'm sure readers' don't do that sort of thing.

```
[1,]    1    3
> # x[[1, ]] is not allowed
>
> df = data.frame(a = 1:2, b = 1:2)
> str(df[1])
'data.frame':   2 obs. of  1 variable:
 $ a: int  1 2
> str(df[[1]])
 int [1:2] 1 2
> str(df[, "a", drop = FALSE])
'data.frame':   2 obs. of  1 variable:
 $ a: int  1 2
> str(df[, "a"])
 int [1:2] 1 2
```

15.4.1 Testing for equality

In addition to the equality operators, the base system includes two equality related functions, `identical` and `all.equal`.

```
> x = 1:5  ;  y = 1:5
> x == y  # Return the result of equality test for each corresponding element
[1] TRUE TRUE TRUE TRUE TRUE
> identical(x, y) # Return a single value denoting exact equality
[1] TRUE
> 1L == 1   # 1L is stored internally as an integer, 1 is stored as a double
[1] TRUE
> identical(1L, 1)  # identical requires the stored type be the same
[1] FALSE
> 0.9 == (1.1 - 0.2)  # could be affected by lack of precision
[1] FALSE
> all.equal(0.9, 1.1 - 0.2)  # do a fuzzy compare
[1] TRUE
> all.equal(0.9, 1.1 - 0.2, tolerance=0)  # find out much how fuzz there is
[1] "Mean relative difference: 1.233581e-16"
```

The default tolerance used by the `all.equal` function is `.Machine$double.eps^0.5`.

Comparisons against NA always returns NA. The `is.na` function can be used to check for this quantity; the anyNA function returns TRUE, if its argument contains at least one NA.

15.4.2 Assignment

Four of the ways of assigning a value to a variable in R include:

```
x <- 3  # Operator used by people who follow the herd
x <<- 3 # Assigns to the x at global scope

3 -> x  # Rarely encountered outside descriptions of the language

x = 3   # Supported since R version 1.4
```

Many R books and articles use the two characters **<-**.[ii] Developers are used to seeing the **=** token, and with nothing other than conformity to existing R usage to recommend the alternative, the assignment token that developers are already very familiar with, is used in this book.

There is one context where **=** does not behave like normal assignment. R supports the use of parameter names in arguments to function calls, to explicitly specify that a named parameter is to be assigned a given value. In the context of a function argument list, the left operand of **=** is treated as the name of a parameter and the right operand as the value to be assigned. An error is flagged, if the function definition does not have a parameter having the specified name.

[ii] The developers of the S language used terminals that had a single key for this symbol sequence.

```
func = function (a, b, c) a + b * c

func(2, 3, 9)
func(c=9, b=3, a=2)

func(d=3, 4, 5)  # no parameter named d, an error is raised

# use <- if the intent is to assign to d and pass this value as an argument
func(d<-3, 4, 5)
```

15.5 The R type (mode) system

R supports values having the following basic types (R also has the concept of *mode*, which is based on semantics rather than underlying representation, e.g., the mode function returns numeric where typeof returns either integer or double):

- NULL:
- raw: essentially uninterpreted byte values,
- logical: holds one of the values: TRUE, FALSE, T or F. The conversion as.logical(any_non_zero_value) returns TRUE,
- character: what many other languages call a string type,
- integer: the only integer type, contains 32 bits (NA is represented using the most negative value, so this value is not available as an integer; trying to generate this, or any other value outside the representable value of a 32-bit integer, will result in a value having a double type),
- double: the only floating-point type, contains 64 bits. Can exactly represent all 32-bit integers,
- complex: contains a real and imaginary double type,

An object may be reported to have one of these basic types, but it may actually be a vector or array of this type.

More complicated types may be created, such as lists, data frames, etc.

15.5.1 Converting the type (mode) of a value

It is often possible to convert the mode (type) of a value by calling the as.some_mode function, where some_mode is the name of a mode, e.g., integer. If a conversion fails, NA is returned.

Conversion precedence

```
NULL < raw < logical < integer < real < complex < character < list < expression
```

15.6 Statements

R contains the usual language constructs that look like statements, but they can behave like expressions:

- **function**: defines a function, whose value has to be assigned to an object:

  f=function(p1, p2) {return(p1+p2)}
- blocks of code are bracketed using the punctuation pair: **{** and **}**,
- **;** (semicolon) is required to delimit multiple expressions on the same line, but is otherwise optional,
- **if**: which takes an optional **else** arm (there is no **then** keyword, but there is an ifthenelse function),
- **for**: which has the form for (i in x), where x is a vector (such as 1:10),
- **while** and **repeat** loops are available,

- loops may be terminated using the **break** keyword or the break function, and may be continued at the next iteration using the **next** keyword or next function,
- return is a function: return(1+return(1)) returns the value 1,
- switch is a function.

15.7 Defining a function

```
> g=1  # a global variable
> f = function(p1, p2) # define a function and assign it to f
+ {
+ l=g  # Value access, check lexical and dynamic scope for g
+ g=2  # Assignment: only check local scope, if no variable exists, create one
+
+ m=h  # h is dynamically in scope
+
+ return(return(1)+1) # return is a function call
+ }
> h=2  # another global variable
> f(1, 2)
[1] 1
> g
[1] 1
> h=3 # At global scope, so must be global variable
```

Argument evaluation is lazy, that is, they are evaluated the first time their value is required.

The ... token (three dots) specifies that a variable number of unknown arguments may be passed.

```
unk_args=function(...)
{
a=list(...) # Convert any arguments passed to a list of values
# Access the list of values in a
}
```

15.8 Commonly used functions

Technically every operation is a function call (so '+'(1, 2) and 1+2 are equivalent), but not all function calls have equivalent operator tokens.

```
> x = 1:10
> if (any(x > 7)) print("At least one value greater than 7")
[1] "At least one value greater than 7"
> if (all(x > 0)) print("All values greater than zero")
[1] "All values greater than zero"
> rep(1:2, 3)
[1] 1 2 1 2 1 2
```

- head/tail mimics the behavior of the Unix head/tail programs,
- length returns the number of elements in its vector argument,
- nrow/ncol return the number of rows/columns in the data frame argument (NROW/NCOL gracefully handle vector arguments),
- order returns a vector containing an index in to the argument in the order needed to sort the argument values,
- str lists the columns in a variable, along with their type and the first few values in each row; it provides a quick way of verifying that columns have the expected type.
- which returns a vector of values containing the index of the argument values that are true,
- methods: list functions overloaded on the argument name
- installed.packages: list all installed packages
- ls: lists variables that exist in the current environment,
- system.time, proc.time: cpu time used, and the real, and cpu time of the currently running R process.

15.9 Input/Output

Functions are available for reading data having a variety of formats (e.g., comma separated values), from all the common data sources (e.g., files, databases, web pages). In some cases the contents of a compressed file will be automatically uncompressed before reading. In the case of files, all the data contained in the file is often read, and returned as a single object.

Many functions try to automatically deduce the datatype of the data read, e.g., whether it is integer, real, character sequence, etc. Sometimes the datatype selected is not correct, and work has to be done to ensure the data is treated as having the desired type; the `read.csv` function bases its decision on the type of each column, by analysing the first 6, or so, lines of the file.

Some functions in the base system, e.g., `read.csv`, convert columns containing string values to factors, by default; the original intent was, presumably, to reduce the storage needed to hold the data. A column of factors, as a type, does not always behave the same as a column of strings and this default conversion behavior is often a liability. Using the argument `as.is=TRUE` prevents values being converted to factors (it is used in all of this book's example code).

```
data=read.csv("measurements.csv.gz", as.is=TRUE) # file will be uncompressed

data=read.csv("measurements.csv", sep="|", as.is=TRUE) # change separator

data=read.csv("https://github.com/Derek-Jones/ESEUR-code-data/blob/master/benchmark/MSI
```

The first line of the input file is assumed to denote the name of each column, specifying `header=FALSE` switches off this default behavior.

All characters on an input line after, and including, the comment character, #, are ignored (various options interact with this behavior, including the `comment.char` option which can be used to change the character used).

The `foreign` package supports the reading (and some writing) of data stored in some of the binary file formats used by other applications (e.g., `read.spss`).

If data is not already in a form that can be easily processed by R, it may be simpler to convert it using a language or tool that you are already familiar with, rather than using R.

The R environment includes a simple spreadsheet like editor for manual data entry and modifying existing data.

```
scores = edit(scores) # invoke built-in spreadsheet like editor
```

There are corresponding `write` functions for many of the `read` functions, e.g., `write.csv`.

The `print` function performs relatively simple formatted output (the `format` function can be used to create more sophisticated formatting, that can then be output); the `cat` function performs relatively little formatting, but is more flexible, and in particular does not terminate its output with a newline; the `sink` function can be used to specify an alternative location to write console output.

15.9.1 Graphical output

There are probably more functions supporting graphical output, in R, than textual output. Perhaps the most commonly used graphical output function is `plot`. This function often does a good job of producing a reasonable graphical representation of the data. Overloaded versions of this function are often provided by packages, to plot data having a particular class created by the package.

By default, graphical output is sent to the console device; this behavior can be overridden to produce a file having a particular format, e.g., `pdf`, `jpeg`, `png` and `pictex`. The list of supported output devices varies across the operating systems on which R runs.

The behavior of the `plot` function can be influenced by previous calls to the `par` function, which set configurable options.

Various packages providing graphical output are available, with the `ggplot` package probably being the most commonly used by frequent R users.

15.10 Non-statistical uses of R

While the target of R's domain specialised functionality is statistical data analysis, there are other application domains where this functionality could be useful (but may not warrant effort needed to learn R).

A variety of functions designed for manipulating the rows and columns of delimited data files are available; see Github–Rlang/Top500.R.

A technique for spotting whether a file contains compressed data (e.g., a virus hidden in a script by compressing it to look like a jumble of numbers) is to plot the fraction of distinct values appearing in successive, fixed size, blocks; see figure 15.2. Compressed data is likely to contain an approximately uniform distribution of byte values (compression is achieved by reducing apparent information content), your mileage may vary between compression methods.

The following code reads a pdf file, applies a sliding window to the data and then plots the fraction of distinct values in each window (at a given offset).

```
window_width=256 # if less than 256, divisor has to change in plot call

plot_unique=function(filename)
{
t=readBin(filename, what="raw", n=1e7)

# Sliding the window over every point is too much overhead
cnt_points=seq(1, length(t)-window_width, 5)

u=sapply(cnt_points, function(X) length(unique(t[X:(X+window_width)])))
plot(u/256, type="l", xlab="Offset", ylab="Fraction Unique", las=1)

return(u)
}

dummy=plot_unique("http://www.coding-guidelines.com/R_code/requirements.tgz")
```

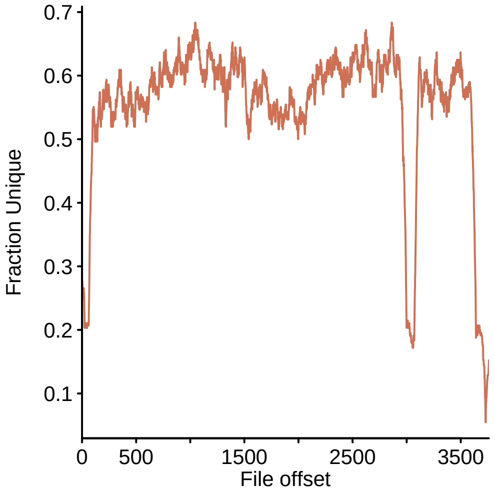


Figure 15.2: The unique bytes per window (256 bytes wide) of a pdf file. Github–Local

15.11 Very large datasets

While most existing software engineering datasets tend to be small, exceptions may occur from time to time. A variety of techniques are available for handling large datasets, including:

- the bigmemory package provides software defined memory management (i.e., swapping data between memory and main storage). The bigtabulate package, along with other big??? packages contain functions that perform commonly used operations on this data.
- the data.table package extends data.frames to support up to 100G of storage,

References

1. 7digital development team statistical analysis report april 2011-2012. blog article, July 2012. http://www.7digital.com. 133, 240, 326, 327, 376
2. J. T. Abbott, J. L. Austerweil, and T. L. Griffiths. Random walks on semantic networks can resemble optimal foraging. *Psychological Review*, 122(3):558–569, July 2015. 31
3. T. Abdel-Hamid and S. E. Madnick. *Software Project Dynamics: An Integrated Approach*. Prentice-Hall, Inc, 1991. 124, 352
4. D. Aboody and B. Lev. The value relevance of intangibles: The case of software capitalization. *Journal of Accounting Research*, 36:161–191, 1998. 80
5. ACAA Technical Agent. Ada conformity assessment test suite (ACATS). website, Jan. 2018. http://www.ada-auth.org/acats.html. 167
6. A. Adamatzky. A brief history of liquid computers. *Philosophical Transactions of The Royal Society B*, 374(1774), June 2019. 1
7. E. N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, Jan. 1984. 153, 155
8. J. Adams. *Risk and Freedom: The record of road safety regulation*. Transport Publishing Projects, 1985. 50
9. B. Adelson. Problem solving and the development of abstract categories in programming languages. *Memory & Cognition*, 9(4):422–433, July 1981. 32, 33
10. ADPE Selection Office. Federal COBOL compiler testing service compiler validation request information. Report No FCCTS/TR-77/05, Department of the Navy, USA, May 1977. 167
11. J. Agar. *The Government Machine A Revolutionary History of the Computer*. The MIT Press, 2003. 88
12. R. Agarwal and M. Gort. The evolution of markets and entry, exit and survival of firms. *The Review of Economics and Statistics*, 78(3):489–498, Aug. 1996. 95
13. P. J. Ågerfalk. Insufficient theoretical contribution: a conclusive rationale for rejection? *European Journal of Information Systems*, 23(6):593–599, Nov. 2014. 6
14. A. Aghayev and P. Desnoyers. Skylight-A window on shingled disk operation. In *13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 135–149, Feb. 2015. 366
15. N. Agrawal. *Representative, reproducible, and practical benchmarking of file and storage systems*. PhD thesis, University of Wisconsin-Madison, 2009. 358
16. N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. *ACM Transactions on Storage*, 5(4):125–138, Dec. 2009. 222
17. N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage*, 3(3):31–45, Oct. 2007. 242
18. M. Ahasanuzzaman, S. Hassan, C.-P. Bezemer, and A. E. Hassan. A longitudinal study of popular ad libraries in the Google Play Store. *Empirical Software Engineering*, 25(1):824–858, Jan. 2020. 89
19. N. Ahmad. Measuring investment in software. OECD Science, Technology and Industry Working Papers 2003/06, OECD, May 2003. 78
20. N. Ahmad, C. Aspden, and OECD Task Force on R&D and Other Intellectual Property Products. *Handbook on Deriving Capital Measures of Intellectual Property Products*. OECD Publishing, 2010. 78
21. J. J. Ahonen and P. Savolainen. Software engineering projects may fail before they are started: Post-mortem analysis of five cancelled projects. *Journal of Systems and Software*, 83(11):2175–2187, Nov. 2010. 116
22. J. J. Ahonen, P. Savolainen, H. Merikoski, and J. Nevalainen. Reported project management effort, project size, and contract type. *The Journal of Systems and Software*, 109(C):205–213, Nov. 2015. 121
23. S. Ajami, Y. Woodbridge, and D. G. Feitelson. Syntax, predicates, idioms – What really affects code complexity? *Empirical Software Engineering*, 24(1):287–328, Feb. 2019. 177
24. F. Akdemir and F. A. Kirmani. Synergy: A synthetic study on teams. Thesis (m.s.), Umeô School of Business, July-Sept. 2008. 76
25. G. A. Akerlof. The market for "Lemons": Quality uncertainty and the market mechanism. *The Quarterly Journal of Economics*, 84(3):488–500, Aug. 1970. 73
26. K. Akita, S. Itagaki, Y. Masawa, M. Nonaka, T. Hatani, K. Hattori, S. Morisaki, Y. Yanagida, T. Takaya, T. Furuyama, and O. Takashi. *Software Development Data White paper 2012-2013*. SEC BOOKS, 2012. 109, 115, 116
27. A. Akshintala, B. Jain, C.-C. Tsai, M. Ferdman, and D. E. Porter. x86-64 instruction usage among C/C++ applications. In *12th ACM International Systems and Storage Conference*, SYSTOR '19, pages 68–79, June 2019. 198
28. H. A. A. Al-Mutawa. On the classification of cyclic dependencies in Java programs. Thesis (m.s.), Massey University, New Zealand, 2013. 206
29. H. Alemzadeh, R. K. Iyer, Z. Kalbarczyk, and J. Raman. Analysis of safety-critical computer failures in medical devices. *IEEE Security & Privacy*, 11(4):14–26, July 2013. 290, 294
30. N. Ali, Z. Sharafi, Y.-G. Guéhéneuc, and G. Antoniol. An empirical study on requirements traceability using eye-tracking. In *28th IEEE International Conference on Software Maintenance*, ICSM'12, pages 191–200, Sept. 2012. 27
31. T. Allee and M. Elsig. Are the contents of international treaties copied-and-pasted? Evidence from preferential trade agreements. Working Paper No. 8, World Trade Institute, Aug. 2016. 77
32. E. J. Allen, P. M. Dechow, D. G. Pope, and G. Wu. Reference-dependent preferences: Evidence from marathon runners. *Management Science*, 63(6):1657–1672, June 2017. 130, 131
33. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architecture*. Morgan Kaufmann Publishers, Mar. 2002. 200
34. R. C. Allen. The British industrial revolution in global perspective: How commerce created the industrial revolution and modern economic growth. Nuffield College, Oxford, 2006. 88
35. T. J. Allen and R. Katz. The dual ladder: Motivational solution or managerial delusion? Working Paper 1692-85, Massachusetts Institute of Technology, Sloan School of Management, Aug. 1985. 104
36. L. Allodi. Economic factors of vulnerability trade and exploitation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS'17, pages 1483–1499, Oct.-Nov. 2017. 150
37. L. Allodi and F. Massacci. A preliminary analysis of vulnerability scores for attacks in wild: The EKITS and SYM datasets. In *Proceedings of the 2012 ACM Workshop on Building analysis datasets and gathering experience returns for security*, DABGERS'12, pages 17–24, Oct. 2012. 150
38. D. A. Almeida, G. C. Murphy, G. Wilson, and M. Hoye. Do software developers understand open source licenses? In *25th IEEE International Conference on Program Comprehension*, ICPC 2017, pages 1–11, May 2017. 65
39. M. G. Almiron, E. S. Almeida, and M. N. Miranda. The reliability of statistical functions in four software packages freely used in numerical computation. *Brazilian Journal of Probability and Statistics*, 23(2):107–119, 2009. 14
40. M. G. Almiron, B. Lopes, A. L. C. Oliveira, A. C. Medeiros, and A. C. Frery. On the numerical accuracy of spreadsheets. *Journal of Statistics*, 34(4):1–29, Apr. 2010. 14
41. A. Almossawi. How maintainable is the Firefox codebase? website, May 2013. http://almossawi.com/firefox/prose. 180
42. W. H. Alsup. ORACLE AMERICA, INC., plaintiff-appellant v. GOOGLE LLC, defendant-cross-appellant. Decision 3:10-cv-03561-WHA, United States District Court for the Northern District of California, Mar. 2018. 109
43. L. E. Alteneder. The learning curve in solving a jig-saw puzzle: A teaching device. *Journal of Educational Psychology*, 26(3):231–232, Mar. 1935. 34
44. E. M. Altmann. *Episodic Memory for External Information*. PhD thesis, Carnegie Mellon University, Aug. 1996. 28

45. E. M. Altmann. Functional decay of memory for tasks. *Psychological Research*, 66(4):287–297, 2002. 24

46. E. M. Altmann, J. G. Trafton, and D. Z. Hambrick. Effects of interruption length on procedural errors. *Journal of Experimental Psychology: Applied*, 23(2):216–229, June 2017. 31

47. H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara. A survival analysis of source files modified by new developers. In *International Conference on Product-Focused Software Process Improvement*, PROFES 2017, pages 80–88, Nov.-Dec. 2017. 159

48. Amazon, Inc. Amazon ec2 service level agreement. https://aws.amazon.com/ec2/sla, June 2013. 370

49. S. Ambler. IT project success survey results. http://www.ambysoft.com/surveys, 2017. 116

50. J. M. Amiri and V. V. K. Padmanabhuni. A comprehensive evaluation of conversion approaches for different function points. Thesis (m.s.), Blekinge Institute of Technology, Sweden, Sept. 2011. 290, 291

51. L. An, O. Mlouki, F. Khomh, and G. Antoniol. Stack Overflow: A code laundering platform? In *eprint arXiv:cs.SE/1703.03897*, Mar. 2017. 77

52. B. C. D. Anda, D. I. K. Sjøberg, and A. Mockus. Variability and reproducibility in software engineering: A study of four companies that developed the same system. *IEEE Transactions on Software Engineering*, 35(3):407–429, May 2009. 116, 126

53. C. Anderson, J. A. D. Hildreth, and L. Howland. Is the desire for status a fundamental human motive? A review of the empirical literature. *Psychological Bulletin*, 141(3):574–601, May 2015. 70

54. D. Anderson. Modeling and analysis of SQL queries in PHP systems. Thesis (m.s.), Department of Computer Science, East Carolina University, Apr. 2018. 198

55. J. R. Anderson. *Learning and Memory: An Integrated Approach*. John Wiley & Sons, Inc, second edition, 2000. 38

56. J. R. Anderson, D. Bothell, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin. An integrated theory of the mind. *Psychological Review*, 111(4):1036–1060, Oct. 2007. 19

57. J. R. Anderson and R. Milson. Human memory: An adaptive perspective. *Psychological Review*, 96(4):703–719, Oct. 1989. 33

58. M. L. Anderson. Neural reuse: A fundamental organizational principle of the brain. *Behavioral and Brain Sciences*, 33(4):245–313, Apr. 2010. 17

59. D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *Proceedings of the 25th USENIX Security Symposium*, SEC'16, pages 583–600, Aug. 2016. 167

60. J. Annett. Subjective rating scales: science or art? *Ergonomics*, 45(12):966–987, 2002. 372

61. A. Ansar. 'AppStore secrets' (What we've learned from 30,000,000 downloads). Presentation, pinch media, 2009. 105

62. F. J. Anscombe. Graphs in statistical analysis. *The American Statistician*, 27(1):17–21, Feb. 1973. 291, 292

63. ANSI X3.9. *American National Standard programming language FORTRAN*. American National Standards Institute, inc., Nov. 1978. 197

64. K. Aoki and M. W. Feldman. Evolution of learning strategies in temporally and spatially variable environments: A review of theory. *Theoretical Population Biology*, 91:3–19, Feb. 2014. 71

65. J. Aranda. Anchoring and adjustment in software estimation. Thesis (m.s.), Graduate Department of Computer Science, University of Toronto, 2005. 123

66. L. Argote, C. A. Insko, N. Yovetich, and A. A. Romero. Group learning curves: The effects of turnover and task complexity on group performance. *Journal of Applied Social Psychology*, 25(6):512–529, Mar. 1995. 73

67. H. R. Arkes, R. M. Dawes, and C. Christensen. Factors influencing the use of a decision rule in a probabilistic task. *Organizational Behavior and Human Decision Processes*, 37:93–110, 1986. 53

68. P. Armer. SHARE – A eulogy to cooperative effort. Technical Report P-969, The RAND Corporation, Oct. 1956. 64, 108

69. J. S. Armstrong. The seer-sucker theory: The value of experts in forecasting. *Technology Review*, pages 16–24, June-July 1980. 37

70. V. Arnaoudova, L. M. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y.-G. Guéhéneuc. REPENT: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, 40(5):502–532, May 2014. 191

71. J. Arndt. *Matters Computational: Ideas, Algorithms, Source Code*. Springer, 2010. 202

72. T. B. Arnold and J. W. Emerson. Nonparametric goodness-of-fit tests for discrete null distributions. *The R Journal*, 3(2):34–39, Dec. 2011. 236

73. A. Arora, S. Belenzon, A. Patacconi, and J. Suh. The changing structure of American innovation: Some cautionary remarks for economic growth. Working Paper No. 25893, National Bureau of Economic Research, Aug. 2019. 8

74. A. Arora and A. Gambardella. *From Underdogs to Tigers: The Rise and Growth of the Software Industry in Brazil, China, India, Ireland, and Israel*. Oxford University Press, Mar. 2005. 58

75. A. Arora, R. Krishnan, R. Telang, and Y. Yang. An empirical analysis of software vendors' patch release behavior: Impact of vulnerability disclosure. *Information Systems Research*, 21(1):115–132, Mar. 2010. 150, 335, 337, 374

76. W. B. Arthur. Competing technologies, increasing returns, and lock-in by historical events. *The Economic Journal*, 99(394):116–131, Mar. 1989. 97

77. W. B. Arthur. *Increasing Returns and Path Dependency in the Economy*. The University of Michigan Press, 1994. 96, 97

78. S. E. Asch. Studies of independence and conformity: A minority of one against a unanimous majority. *Psychological Monographs: General and Applied*, 70(9):1–70, 1956. 52

79. A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. A survey on compiler autotuning using machine learning. In *eprint arXiv:cs.PL/1801.04405*, Mar. 2018. 174

80. T. A. Åstebro, S. A. Jeffrey, and G. K. Adomdza. Inventor perseverance after being told to quit: The role of cognitive biases. *Journal of Behavioral Decision Making*, 20(3):253–272, Apr. 2007. 53

81. S. Atkinson and G. Benefield. Software development: Why the traditional contract model is not fit for purpose. In *46th Hawaii International Conference on System Sciences*, HICSS, pages 4842–4851, Jan. 2013. 120

82. V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh. POSIX abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys'16, page 19, Apr. 2016. 111, 112

83. Audit Scotland. i6: a review. Report, Audit Scotland, Mar. 2017. 116

84. Auerbach. Auerbach guide to time sharing. Computer technology report, Auerbach Publishers Inc., Jan. 1973. 101

85. N. R. Augustine. *Augustine's Laws*. American Institute of Aeronautics and Astronautics, Inc, sixth edition, 1997. 172

86. R. Auler and E. Borin. A LLVM just-in-time compilation cost analysis. Technical Report IC-13-13, Instituto de Computação Universidade Estadual de Campinas, May 2013. 176, 177

87. P. C. Austin. A tutorial on multilevel survival analysis: Methods, models and applications. *International Statistical Review*, 85(2):185–203, Aug. 2017. 336

88. R. D. Austin. The effects of time pressure on quality in software development: An agency model. *Information Systems Research*, 12(2):195–207, June 2001. 74, 75

89. AUTOSAR. *Guidelines for the use of the C++14 language in critical and safety-related systems*. AUTOSAR, 839 edition, Mar. 2017. 149

90. J. L. Autran, D. Munteanu, P. Roche, and G. Gasiot. Real-time soft-error rate measurements: A review. *Microelectronics Reliability*, 54(8):1455–1476, Aug. 2014. 162

91. J.-L. Autran, S. Semikh, D. Munteanu, S. Serre, G. Gasiot, and P. Roche. Soft-error rate of advanced SRAM memories: Modeling and monte carlo simulation. In M. Andriychuk, editor, *Numerical Simulation – From Theory to Industry*, chapter 15, pages 309–336. InTech, Sept. 2012. 162

92. G. Avelino, L. Passos, A. Hora, and M. T. Valente. Measuring and analyzing code authorship in 1+118 open source projects. *Science of Computer Programming*, 176:14–32, May 2019. 137

93. E. Avidan. The significance of method parameters and local variables as beacons for comprehension: An empirical study. Thesis (m.s.), The Hebrew University of Jerusalem, Nov. 2016. 191

94. P. Azoulay, C. Fons-Rosen, and J. S. G. Zivin. Does science advance one funeral at a time? Working Paper No. 21788, National Bureau of Economic Research, USA, Dec. 2015. 9

95. R. H. Baayen, P. Milin, and M. Ramscar. Frequency in lexical processing. *Aphasiology*, 30(11):1174–1220, Mar. 2016. 190

96. C. Babbage, ESQ. *Reflections on the Decline of Science in England, and on Some of its Causes*. B. Fellows, Ludgate Street; and J. Booth, Duke Street, 1830. 10

97. V. Babka. *Improving Accuracy of Software Performance Models on Multicore Platforms with Shared Caches*. PhD thesis, Faculty of Mathematics and Physics, Charles University in Prague, Oct. 2012. 368

98. V. Babka and P. Tůma. Investigating cache parameters of x86 family processors. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 77–96, Jan. 2009. 368, 369

99. D. Baccarini, G. Salm, and P. E. D. Love. Management of risks in information technology projects. *Industrial Management & Data Systems*, 104(4):286–295, 2004. 126

100. A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE'13, pages 712–721, May 2013. 165

101. A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: Bugs and bug-fix commits. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2010, pages 97–106, Nov. 2010. 148

102. A. Back and E. Westman. Comparing programming languages in Google code jam. Thesis (m.s.), Department of Computer Science and Engineering, Chalmers University of Technology, 2017. 192

103. J. Backus. The history of FORTRAN I, II, and III. *SIGPLAN Notices*, 13(8):165–180, 1978. 108

104. J. Backus. Programming in America in the 1950s– some personal impressions. In N. Metropolis, J. Howlett, and G.-C. Rota, editors, *A History of Computing in the Twentieth Century*, pages 125–135. Academic Press, Feb. 1981. 108, 109

105. J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, H. L. Herrick, R. A. Hughes, L. B. Mitchell, R. A. Nelson, R. Nutt, D. Sayre, P. B. Sheridan, H. Stern, and I. Ziller. *The FORTRAN Automatic Coding System for the IBM 704 EDPM: Programmer's Reference Manual*. International Business Machines Corporation, 590 Madison Avenue, New York 22, N.Y., Oct. 1956. 109

106. A. Bacon, S. Handley, and S. Newstead. Individual differences in strategies for syllogistic reasoning. *Thinking & Reasoning*, 9(2):133–168, 2003. 43

107. A. Baddeley. Working memory. In A. Baddeley, M. W. Eysenck, and M. Anderson, editors, *Memory*, chapter 3, pages 41–69. Psychology Press, Feb. 2009. 29

108. A. Baddeley. Working memory: Theories, models, and controversies. *Annual Review of Psychology*, 63:1–29, Sept. 2012. 29

109. A. D. Baddeley, N. Thomson, and M. Buchanan. Word length and the structure of short-term memory. *Journal of Verbal Learning and Verbal Behavior*, 14(6):575–589, Dec. 1975. 29, 358

110. M. Bagherzadeh, N. Kahani, C.-P. Bezemer, A. E. Hassan, J. Dingel, and J. R. Cordy. Analyzing a decade of Linux system calls. *Empirical Software Engineering*, 23(3):1519–1551, June 2018. 112

111. J. N. Bailenson, M. S. Shum, S. Atran, D. L. Medin, and J. D. Coley. A bird's eye view: biological categorization and reasoning within and across cultures. *Cognition*, 84:1–53, 2002. 41

112. D. H. Bailey. Misleading performance reporting in the supercomputer field. Technical Report RNR-92-005, Numerical Aerodynamic Simulation Division, NASA Ames Research Center, Dec. 1992. 362

113. S. Baily, R. Gilbertson, and E. Straub. Modular multimode radar (CMMR) software acquisition study. Technical Report 2302-01-1-2291, ARINC Research Corporation, Mar. 1981. 103

114. E. Bainomugisha, A. L. Carreton, T. van Cutsem, S. Mostinckx, and W. de Meuter. A survey on reactive programming. *ACM Computing Surveys*, 45(4):52, Aug. 2013. 175

115. P. Bajari, S. Tadelis, and S. Houghton. Bidding for incomplete contracts: An empirical analysis of adaptation costs. *American Economic Review*, 104(4):1288–1319, Oct. 2011. 119

116. S. S. Bajwa, X. Wang, A. N. Duc, and P. Abrahamsson. Failures to be celebrated: an analysis of major pivots of software startups. In *eprint arXiv:cs.SE/1710.04037*, Oct. 2017. 126

117. A. H. Baker, D. M. Hammerling, M. N. Levy, H. Xu, J. M. Dennis, B. E. Eaton, J. Edwards, C. Hannay, S. A. Mickelson, R. B. Neale, D. Nychka, J. Shollenberger, J. Tribbia, M. Vertenstein, and D. Williamson. A new ensemble-based consistency test for the Community Earth System Model (pyCECT v1.0). *Geoscientific Model Development*, 8:2829–2840, Sept. 2015. 147

118. F. T. Baker. Chief programmer team management of production programming. *IBM Systems Journal*, 11(1):56–73, 1972. 69, 137

119. M. Bakkaloglu, J. J. Wylie, C. Wang, and G. R. Ganger. On correlated failures in survivable storage systems. Technical Report CMU-CS-02-129, Carnegie Mellon University, May 2002. 274

120. B. Balaji, J. McCullough, R. K. Gupta, and Y. Agarwal. Accurate characterization of the variability in power consumption in modern mobile processors. In *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, HotPower'12, Oct. 2012. 317

121. M. Baldwin. Scientific autonomy, public accountability, and the rise of "peer review" in the Cold war United States. *Isis*, 109(3):538–558, Sept. 2018. 9

122. T. Ball and J. R. Larus. Branch prediction for free. Technical Report #1137, Computer Sciences Department, University of Wisconsin–Madison, Feb. 1993. 198

123. S. Baltes and S. Diehl. Usage and attribution of Stack Overflow code snippets in GitHub projects. In *eprint arXiv:cs.SE/1802.02938*, Feb. 2018. 78

124. S. Baltes and P. Ralph. Sampling in software engineering research: A critical review and guidelines. *ACM Transactions on Software Engineering and Methodology*, ???(???):???, Apr. 2020. 6

125. N. Banerjee, A. Rahmati, M. D. Corner, S. Rollins, and L. Zhong. Users and batteries: Interactions and adaptive energy management in mobile systems. In *International Conference on Ubiquitous Computing*, UbiComp 2007, pages 217–237, Sept. 2007. 91

126. P. Banyard and N. Hunt. Something missing? *The Psychologist*, 13(2):68–71, 2000. 19

127. L. Bao, Z. Xing, X. Xia, D. Lo, and S. Li. Who will leave the company?: A large-scale industry study of developer turnover by mining monthly work report. In *14th IEEE/ACM International Conference on Mining Software Repositories*, MSR'17, pages 170–181, May 2017. 137

128. J. H. Barkow, L. Cosmides, and J. Tooby. *The Adapted Mind: Evolutionary Psychology and the Generation of Culture*. Oxford University Press, 1992. 18

129. W. P. Barnett. *The Red Queen among Organizations: How competitiveness evolves*. Princeton University Press, 2008. 2, 91

130. A. Baronchelli, V. Loreto, and A. Puglisi. Individual biases, cultural evolution, and the statistical nature of language universals: The case of colour naming systems. *PLoS ONE*, 10(5):e0125019, May 2015. 197

131. D. R. Barrett. *World Christian Encyclopedia: A Comparative Survey of Churches and Religions in the Modern World AD 1900-2000*. Oxford University Press, 1982. 92

132. E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt. Virtual machine warmup blows hot and cold. In *eprint arXiv:cs.PL/1602.00602v4*, July 2017. 356

133. L. Barrett, R. Dunbar, and J. Lycett. *Human Evolutionary Psychology*. Palgrave Macmillan, 2002. 18

134. L. A. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. Report, Morgan & Claypool, 2009. 91

135. V. R. Basili and J. Beane. Can the Parr curve help with manpower distribution and resource estimation problems? *The Journal of Systems and Software*, 2(1):59–69, Feb. 1981. 124

136. V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. V. Zelkowitz. The empirical investigation of perspective-based reading. In *Proceedings of the Twentieth Annual Software Engineering Workshop*, pages 21–69, Dec. 1995. 6, 355

137. V. R. Basili, N. M. Panlilio-Yap, C. L. Ramsey, C. Shih, and E. E. Katz. A quantitative analysis of software developed in Ada. Technical Report TR-1403, Department of Computer Science, University of Maryland, May 1984. 47

138. V. R. Basili and A. J. Turner. Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering*, SE-1(4):390–396, Dec. 1975. 128

139. F. M. Bass. A new product growth model for consumer durables. *Management Science*, 15(5):215–227, Jan. 1969. 82

140. P. I. Bass and F. M. Bass. Diffusion of technology generations: A model of adoption and repeat sales. website, 2001. www.bassbasement.org/F/N/FMB/Pubs/Bass and Bass 2001.pdf. 82

141. H. A. Bastiaanse. *Very, Many, Small, Penguins: Vaguely Related Topics*. PhD thesis, Institute for Logic, Language and Computation, Universiteit van Amsterdam, Mar. 2014. 47

142. B. Baudry, S. Allier, and M. Monperrus. Tailored source code transformations to synthesize computationally diverse program variants. In *eprint arXiv:cs.SE/1401.7635v1*, Jan. 2014. 159, 195

143. F. L. Bauer and H. Wössner. The "Plankalkül" of Konrad Zuse: a forerunner of today's programming languages. *Communications of the ACM*, 15(7):678–685, July 1972. 109

144. J. Bauer, J. Siegmund, N. Peitek, J. C. Hofmeister, and S. Apel. Indentation: Simply a matter of style or support for program comprehension? In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC'19, pages 154–164, May 2019. 187

145. A. Baumann. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS'17, pages 132–137, May 2016. 114

146. R. F. Baumeister. *Is There Anything Good About Men?* Oxford University Press, 2010. 19

147. R. T. Baust. *Computer Characteristics Quarterly: Volume 7, Number 4-Volume 8, Number 1*. adams associates, 1968. 90

148. G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. The evolution of project inter-dependencies in a software ecosystem: the case of Apache. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM'13, pages 280–289, Sept. 2013. 98

149. G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. How the Apache community upgrades dependencies: An evolutionary study? *Empirical Software Engineering*, 20(5):1275–1317, Oct. 2015. 98, 99

150. O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. The influence of non-technical factors on code review. In *20th Working Conference on Reverse Engineering*, WCRE 2013, pages 122–131, Oct. 2013. 140, 141

151. B. L. Bayus, S. Jain, and A. G. Rao. Truth or consequences: An analysis of vaporware and new product announcements. *Journal of Marketing Research*, 38(1):3–13, Feb. 2001. 73

152. A. A. Beaujean. *Latent Variable Modeling Using R*. Routledge, 2014. 371

153. B. Beber and A. Scacco. What the numbers say: A digit-based test for election fraud. *Political Analysis*, 20(2):211–234, Apr. 2012. 382

154. C. Becker, F. Fagerholm, R. Mohanani, and A. Chatzigeorgiou. Temporal discounting in technical debt: How do software practitioners discount the future? In *eprint arXiv:cs.SE/1901.07024*, Jan. 2019. 54

155. G. S. Becker. Investment in human capital: A theoretical analysis. *Journal of Political Economy*, 70(5):9–49, Oct. 1962. 67

156. R. A. Becker and W. S. Cleveland. *Trellis Graphics User's Manual*. AT&T Bell Laboratories, Murray Hill, Dec. 1995. 221

157. J. Beckhusen. Occupations in information technology. American Community Survey Report ACS-35, U.S. Census Bureau, Aug. 2016. 104

158. M. Bekoff, C. Allen, and G. M. Burghardt. *The Cognitive Animal: Empirical and Theoretical Perspectives on Animal Cognition*. MIT Press, 2002. 18

159. R. W. Belk and G. Tumbat. The cult of Macintosh. *Consumption, Markets and Culture*, 8(3):205–217, Sept. 2005. 70

160. C. G. Bell. Fundamentals of time shared computers. *Computer Design*, 7(2):44–59, Feb. 1968. 1

161. C. G. Bell. The mini and micro industries. *Computer*, 17(10):14–30, Oct. 1984. 90

162. G. Bell. Bell's law for the birth and death of computer classes: A theory of the computer's evolution. MSR-TR 2007-146, Microsoft Research, Silicon Valley, Nov. 2007. 90

163. G. Bell. Supercomputers: The amazing race (A history of supercomputing, 1960-2020). Technical Report MSR-TR-2015-2, Microsoft Research, Nov. 2014. 90

164. V. A. Bell and P. N. Johnson-Laird. A model theory of modal reasoning. *Cognitive Science*, 22(1):25–51, 1998. 42

165. M. Beller, A. Zaidman, A. Karpov, and R. A. Zwaan. The last line effect explained. *Empirical Software Engineering*, 22(3):1508–1536, June 2017. 77, 159

166. D. J. Bem. Feeling the future: Experimental evidence for anomalous retroactive influences on cognition and affect. *Journal of Personality and Social Psychology*, 100(3):407–425, Mar. 2011. 261

167. R. W. Bemer. a view of the history of COBOL. *Honeywell Computer Journal*, 5(3):130–135, Nov. 1959. 108

168. O. A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 spot instance pricing. *ACM Transactions on Economics and Computation*, 1(3), Sept. 2013. 58

169. G. Beniamini, S. Gingichashvili, A. K. Orbach, and D. G. Feitelson. Meaningful identifier names: The case of single-letter variables. In *25th IEEE International Conference on Program Comprehension*, ICPC 2017, pages 45–54, May 2017. 190

170. J. R. Beniger. *The Control Revolution: Technological and Economic Origins of the Information Society*. Hardvard University Press, 1986. 3

171. Y. Benkler. Coase's Penguin, or, Linux and the nature of the firm. *The Yale Law Journal*, 112(3), Dec. 2002. 57, 64

172. A. Benson, D. Li, and K. Shue. Promotions and the Peter principle. Working Paper n. 3047193, US universities, Feb. 2018. 104

173. R. A. Bentley, C. P. Lipo, H. A. Herzog, and M. W. Hahn. Regular rates of popular culture change reflect random copying. *Evolution and Human Behavior*, 28(3):151–158, May 2007. 72

174. F. C. Y. Benureau and N. P. Rougier. Re-run, repeat, reproduce, reuse, replicate: Transforming code into scientific contributions. In *eprint arXiv:cs.GL/1708.08205*, Aug. 2017. 109

175. E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek, and J. Vitek. On the impact of programming languages on code quality: A reproduction study. *ACM Transactions on Programming Languages and Systems*, 41(4):21, Nov. 2019. 11

176. T. Berger, S. She, K. Czarnecki, and A. Wąsowski. Feature-to-code mapping in two large product lines. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 498–499. Springer Berlin Heidelberg, 2010. 237

177. T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability modeling in the systems software domain. Technical Report GSDLAB-TR 2012-07-06, Generative Software Development Laboratory, University of Waterloo, July 2012. 135

178. E. Berghout, M. Nijland, and K. Grant. Seven ways to get your favoured IT project accepted – politics in IT evaluation. *The Electronic Journal of Information Systems Evaluation*, 8(1):31–40, 2005. 118

179. M. Berglund, W. Bester, and B. van der Merwe. Formalising Boost POSIX regular expression matching. In *International Colloquium on Theoretical Aspects of Computing*, ICTAC 2018, pages 99–115, Oct. 2018. 168

180. B. Berlin and P. Kay. *Basic Color Terms: Their Universality and Evolution*. Berkeley: University of California Press, 1969. 197

181. R. Berman, L. Pekelis, A. Scott, and C. Van den Bulte. p-hacking and false discovery in A/B testing. Working Paper n. 3204791, US universities, Dec. 2018. 359

182. D. Bermbach and E. Wittern. Benchmarking web API quality. In *International Conference on Web Engineering*, ICWE'16, pages 188–206, June 2016. 163

183. A. Bernardo and I. Welch. On the evolution of overconfidence and entrepreneurs. *Journal of Economics & Management Strategy*, 10(3):301–330, 2001. 69

184. K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer. High-performance CMOS variability in the 65-nm regime and beyond. *IBM Journal of Research and Development*, 50(4/5):433–449, July 2006. 362

185. D. M. Berry, K. Daudjee, J. Dong, I. Fainchtein, M. A. Nelson, T. Nelson, and L. Ou. User's manual as a requirements specification: Case studies. *Requirements Engineering*, 9(1):67–82, Feb. 2004. 132

186. D. M. Berry, E. Kamsties, and M. M. Krieger. From contract drafting to software specification: Linguistic sources of ambiguity. A Handbook, Nov. 2003. 157

187. L. M. A. Bettencourt, A. Cintrón-Arias, D. I. Kaiserd, and C. Castillo-Chávez. The power of a good idea: Quantitative modeling of the spread of ideas from epidemiological models. *Physica A*, 364:513–536, May 2006. 72

188. K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In C. Beeri and P. Buneman, editors, *Database Theory: 7th International Conference*, ICDT'99, pages 217–235. Springer-Verlag, Jan. 1999. 345

189. D. Biber, S. Johansson, G. Leech, S. Conrad, and E. Finegan. *Longman Grammar of Spoken and Written English*. Pearson Education, 1999. 43, 157, 195

190. C. Bicchieri. *The Grammar of Society: The Nature and Dynamics of Social Norms*. Cambridge University Press, Mar. 2006. 68, 74

191. B. Biddle, A. White, and S. Woods. How many standards in a laptop? (and other empirical questions). Working Paper n. 1619440, Arizona State University (ASU) - College of Law, Sept. 2010. 77

192. S. Biddle. Like everyone else, Twitter hides from U.S. taxes in Ireland. website, Oct. 2013. http://valleywag.gawker.com/like-everyone-else-twitter-hides-from-u-s-taxes-in-ir-1447085830. 80

193. B. Biegel, F. Beck, W. Hornig, and S. Diehl. The order of things: How developers sort fields and methods. In *28th IEEE International Conference on Software Maintenance*, ICSM'12, pages 88–97, Sept. 2012. 205, 206, 351

194. S. Bikhchandani, D. Hirshleifer, and I. Welch. A theory of fads, fashion, custom, and cultural change as informational cascades. *Journal of Political Economy*, 100(5):992–1026, Oct. 1992. 52

195. P. Bilton, P. Dodimead, E. Livingstone, I. Rayner, G. Turner, M. Wynniatt, and S. Howes. Managing the risks of legacy ICT to public service delivery. HC 539 SESSION 2013-14, National Audit Office, UK, Sept. 2013. 91, 140

196. W. L. Bircher. *Predictive Power Management for Multi-Core Processors*. PhD thesis, The University of Texas at Austin, Dec. 2010. 364, 367

197. C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced? Bias in bug-fix datasets. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, FSE 2009, pages 121–130, Aug. 2009. 148

198. S. Bird. Software knows best: A case for hardware transparency and measurability. Thesis (m.s.), Department of Electrical Engineering and Computer Science, University of California at Berkeley, May 2010. 358

199. P. G. Bishop and R. E. Bloomfield. Worst case reliability prediction based on a prior estimate of residual defects. In *Proceedings 13th International Symposium on Software Reliability Engineering*, ISSRE'02, pages 295–303, Nov. 2002. 155

200. T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *37th Annual International Computer Software & Applications Conference*, COMPSAC 2013, pages 303–312, July 2013. 137, 218

201. Bitsavers' pdf document archive. Document archive: website, July 2019. http:bitsavers.trailing-edge.com/pdf. 111

202. E. Biyalagorsky, W. Boulding, and R. Staelin. Stuck in the past: Why managers persist with new product failures. *Journal of Marketing*, 70(2):108–121, Apr. 2006. 56, 130

203. N. M. Blachman. A survey of automatic digital computers. Survey 111293, Office of Naval Research, Washington, D.C., 1953. 108, 361

204. S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis (extended version). Technical Report TR-CS-06-01, Department of Computer Science, Australian National University, Aug. 2006. 267

205. A.-R. Blais and E. U. Weber. A domain-specific risk-taking (DOSPERT) scale for adult populations. *Judgment and Decision Making*, 1(1):33–47, Apr. 2006. 51

206. D. M. Blank and G. J. Stigler. *The Demand and Supply of Scientific Personnel*. National Bureau of Economic Research, Inc., 1957. 104, 129

207. M. S. Blaubergs and M. D. S. Braine. Short-term memory limitations on decoding self-embedded sentences. *Journal of Experimental Psychology*, 102(4):745–748, 1974. 30

208. D. S. Blinder and D. M. Oppenheimer. Beliefs about what types of mechanisms produce random sequences. *Journal of Behavioral Decision Making*, 21(4):414–427, Oct. 2008. 382

209. N. Bloom, T. Kretschmer, and J. van Reenen. Are family-friendly workplace practices a valuable firm resource? *Strategic Management Journal*, 32(4):343–367, Apr. 2011. 105

210. B. I. Blum. Improving software maintenance by learning from the past: A case study. *Proceedings of the IEEE*, 77(4):596–606, Apr. 1989. 142

211. J. Boccara. Good news: strong types are (mostly) free in C++. website, May 2017. http://www.fluentcpp.com/2017/05/05/news-strong-types-are-free. 201

212. B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Inc, 1981. 291

213. B. W. Boehm and P. N. Papaccio. A value-chain analysis or software productivity. Technical Report USC-CSE-86-500, Center for Systems and Software Engineering, University of Southern California, 1986. 81

214. G. D. Boetticher. Improving credibility of machine learner models in software engineering. In D. Zhang and J. J. P. Tsai, editors, *Advances in Machine Learning Applications in Software Engineering*, chapter 3, pages 52–73. Idea Group Publishing, Oct. 2006. 375

215. J. G. Bolten, R. S. Leonard, M. V. Arena, O. Younossi, and J. M. Sollinger. Sources of weapon system cost growth: Analysis of 35 major defense acquisition programs. Monograph series, RAND Corporation, 2008. 121

216. C. F. Bond, Jr. and L. J. Titus. Social facilitation: A meta-analysis of 241 studies. *Psychological Bulletin*, 94(2):265–292, Sept. 1983. 76

217. J. Bonvoisin, R. Mies, J.-F. Boujut, and R. Stark. What is the "source" of open source hardware? *Journal of open hardware*, 1(1):5, Sept. 2017. 66

218. C. F. Borges. An improved algorithm for HYPOT(A,B). In *eprint arXiv:math.NA/1904.09481*, June 2019. 146

219. R. Bornat, S. Dehnadi, and S. ??? Mental models, consistency and programming aptitude. In *Tenth Australasian Computing Education Conference*, ACE'08, pages 53–61, Jan. 2008. 175

220. L. Boroditsky. Metaphoric structuring: understanding time through spatial metaphors. *Cognition*, 75:1–28, 2000. 102

221. A. Börsch-Supan and M. Weiss. Productivity and age: Evidence from work teams at the assembly line. Technical Report 148-2007, Manheim Research Institute for the Economics of Aging, 2007. 56

222. L. Bossavit. *The Leprechauns of Software Engineering: How folklore turns into fact and what to do about it*. Leanpub, 2016. 79

223. N. Bostrom and A. Sandberg. The wisdom of nature: An evolutionary heuristic for human enhancement. In J. Savulescu and N. Bostrom, editors, *Human Enhancement*, chapter 18, pages 375–416. Oxford University Press, Jan. 2011. 18

224. A. Botchkarev. Estimating the accuracy of the return on investment (ROI) performance evaluations. *Interdisciplinary Journal of Information, Knowledge, and Management*, 10:217–233, 2015. 59

225. L. Boué. Real numbers, data science and chaos: How to fit any dataset with a single parameter. In *eprint arXiv:cs.LG/1904.12320*, Apr. 2019. 277

226. K. Boukhetala and A. Guidoum. Sim.DiffProc: A package for simulation of diffusion processes in R. HAL Id: hal-00629841, HAL archives-ouvertes.fr, Oct. 2011. 331

227. J. Bourn. New IT systems for Magistrates' courts: the Libra project. Report by the Comptroller and Auditor General HC 327 Session 2002-2003, National Audit Office, UK, Jan. 2003. 119, 120

228. E. M. Bowden and M. Jung-Beeman. Normative data for 144 compound remote associate problems. *Behavior Research Methods, Instruments, & Computers*, 35(4):634–639, Dec. 2003. 76

229. G. H. Bower, J. B. Black, and T. J. Turner. Scripts in memory for text. *Cognitive Psychology*, 11(2):177–220, Apr. 1979. 184

230. J. S. Bowers and C. J. Davis. Bayesian just-so stories in psychology and neuroscience. *Psychological Bulletin*, 138(3):389–414, 2012. 19, 250

231. R. Boyd and P. J. Richerson. Why does culture increase human adaptability. *Ethology and Sociobiology*, 16(2):125–143, Mar. 1995. 71

232. R. Boyd and P. J. Richerson. Why culture is common, but cultural evolution is rare. *Proceedings of the British Academy*, 88:77–93, Apr. 1996. 69

233. M. G. Bradac, D. E. Perry, and L. G. Votta. Prototyping a process monitoring experiment. *IEEE Transactions on Software Engineering*, 20(10):774–784, 1994. 129, 130

234. T. F. Brady, T. Konkle, G. A. Alvarez, and A. Oliva. Visual long-term memory has a massive storage capacity for object details. *PNAS*, 105(38):14325–14329, Sept. 2008. 55

235. D. Braha and Y. Bar-Yam. The statistical mechanics of complex product development: Empirical and analytical results. *Management Science*, 53(7):1127–1145, July 2007. 175

236. D. W. Braithwaite and R. L. Goldstone. Flexibility in data interpretation: effects of representational format. *frontiers in Psychology*, 4(980):1–16, Dec. 2013. 220

237. N. R. Bramley. *Constructing the world: Active causal learning in cognition*. PhD thesis, University College London, Feb. 2017. 45

238. M. C. Branco, Y. Xiong, K. Czarnecki, J. Küster, and H. Völzer. An empirical study on consistency management of business and IT process models. Technical Report GSDLAB-TR 2012-03-02, Generative Software Development Laboratory, University of Waterloo, Mar. 2012. 98

239. S. Brand. *How buildings Learn: What happens after they're built*. Viking, 1994. 140

240. J. D. Bransford and J. J. Franks. The abstraction of linguistic ideas. *Cognitive Psychology*, 2(4):331–350, Oct. 1971. 183, 184

241. J. D. Bransford and M. K. Johnson. Contextual prerequisites for understanding: Some investigations of comprehension and recall. *Journal of Verbal Learning and Verbal Behavior*, 11(6):717–726, Dec. 1972. 182

242. G. Branwen. Laws of tech: Commoditize your complement. blog: Gwern, Mar. 2018. `http://www.gwern.net/Complement`. 85

243. S. Brass and C. Goldberg. Semantic errors in SQL queries: A quite complete list. *Journal of Systems and Software*, 79(5):630–644, May 2006. 178

244. H. Braveman. *Labor and Monopoly Capital: The Degradation of Work in the Twentieth Century*. Monthly Review Press, Jan. 1974. 69

245. R. A. Brealey, S. C. Myers, and F. Allen. *Principles of Corporate Finance*. McGraw-Hill Irwin, 10th edition, 2011. 60, 78

246. B. Brembs, K. Button, and M. Munafò. Deep impact: Unintended consequences of journal rank. *Frontiers in Human Neuroscience*, 7(291), June 2013. 9

247. S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information needs in bug reports: Improving cooperation between developers and users. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, CSCW'10, pages 301–310, Feb. 2010. 217

248. C. A. Brewer. Color use guidelines for mapping and visualization. In A. M. Maceachren and D. R. F. Taylor, editors, *Visualization in Modern Cartography*, chapter 7, pages 123–147. Pergamon, Nov. 1994. 224

249. E. Brewer, L. Ying, L. Greenfield, R. Cypher, and T. Ts'o. Disks for data centers. Technical report, Google, Inc, Feb. 2016. 366

250. Brigham Young trace repository. No longer available: website, 201? Copy kindly supplied by Dror G. Feitelson. 153

251. P. Brinch Hansen and R. House. The COBOL compiler for the Siemens 3003. *BIT*, 6(1):1–23, Mar. 1966. 109

252. S. Broadbent. Font requirements for next generation air traffic management systems. Technical Report HRS/HSP-006-REP-01, European Organisation for the Safety of Air Navigation, 2000. 27

253. G. W. Brock. *The U.S. Computer Industry: A Study of Market Power*. Ballinger Publishing Company, 1975. 88

254. L. D. Brock and H. A. Goodman. Reliability analysis of the F-8 digital fly-by-wire system. NASA Contractor Report 163110, Dryden Flight Research Center, Oct. 1981. 143

255. A. D. Broido and A. Clauset. Scale-free networks are rare. In *eprint arXiv:physics.soc-ph/1801.03400*, Jan. 2018. 235

256. G. Bronevetsky and B. R. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS'08, pages 155–164, June 2008. 163

257. J. Brooke. SUS: A 'quick' and 'dirty' usability scale. In P. W. Jordan, B. Thomas, B. A. Weerdmeester, and I. L. McClelland, editors, *Usability Evaluation in Industry*, chapter 21, pages 189–194. Taylor and Francis, June 1996. 372

258. J. Brooke. SUS: A retrospective. *Journal of Usability Studies*, 8(2):29–40, Feb. 2013. 372

259. R. Brooks. A model of human cognitive behavior in writing code for computer programs, vol I. Report AFOSR-TR-75-1084, Carnegie Mellon University, May 1975. 35

260. F. P. Brooks, Jr. *The Mythical Man-Month*. Addison–Wesley, anniversary edition, 1995. 6, 138

261. G. D. A. Brown, I. Neath, and N. Chater. A temporal ratio model of memory. *Psychological Review*, 114(3):539–576, July 2007. 31, 32

262. N. C. C. Brown and A. Altadmri. Novice Java programming mistakes: Large-scale data vs. educator beliefs. *ACM Transactions on Computing Education*, 17(2):7, June 2017. 157

263. J. Brunner and P. C. Austin. Inflation of Type I error rate in multiple regression when independent variables are measured with error. *The Canadian Journal of Statistics*, 37(1):33–46, Mar. 2009. 284

264. M. Brysbaert, W. Fias, and M.-P. Noël. The Whorfian hypothesis and numerical cognition: is 'twenty-four' processed in the same way as 'four-and-twenty'? *Cognition*, 66(1):51–77, Apr. 1998. 197

265. I. Buchmann. *Batteries in a Portable World: A Handbook on rechargeable Batteries for Non-engineers*. Cadex Electronix Inc, third edition, 2011. 364

266. J. B. Buckheit and D. L. Donoho. WaveLab and reproducible research. In A. Antoniadis and G. Oppenheim, editors, *Wavelets and Statistics*, chapter 5, pages 55–81. Springer-Verlag, 1995. 9

267. M. Budden, P. Hadavas, L. Hoffman, and C. Pretz. Generating valid 4 × 4 correlation matrices. *Applied Mathematics E-Notes*, 7:53–59, 2007. 231

268. D. V. Budescu, H.-H. Por, S. B. Broomell, and M. Smithson. The interpretation of IPCC probabilistic statements around the world. *Nature Climate Change*, 4:508–512, Apr. 2014. 149

269. D. J. Buettner. *Designing an Optimal Software Intensive System Acquisition: A Game Theoretic Approach*. PhD thesis, University of Southern California, Sept. 2008. 124, 131, 138, 323, 324, 352, 379

270. E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing virtualization to the x86 architecture with the original VMware workstation. *ACM Transactions on Computer Systems*, 30(4):12, Nov. 2012. 118

271. M. Bullynck. What is an operating system? A historical investigation (1954–1964). HAL Id: halshs-01541602, HAL archives-ouvertes.fr, Aug. 2017. 108

272. N. Bulnet and M. H. Halstead. Impurities found in algorithm implementations. Technical Report CSD-TR 111, Purdue University, Mar. 1974. 194

273. J. S. Bunderson and K. M. Sutcliffe. Management team learning orientation and business unit performance. *Journal of Applied Psychology*, 88(3):552–560, June 2003. 72

274. Bureau of labor statistics. website, July 2019. `https://www.bls.gov/ces`. 99

275. K. P. Burnham and D. R. Anderson. Multimodel inference: Understanding AIC and BIC in model selection. *Sociological Methods and Research*, 33(2):261–304, Nov. 2004. 286

276. J. C. Burns. The evolving market for word processing and typesetting systems. In *Proceedings of the National Computer Conference and Exposition*, AFIPS'76, pages 617–623, June 1976. 88

277. Q. L. Burrell. A note on ageing in a library circulation model. *Journal of Documentation*, 41(2):100–115, 1985. 33

278. R. P. L. Buse and W. R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, July 2010. 188

279. J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins*. PhD thesis, Eindhoven University of Technology, Sept. 2013. 333, 334

280. R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19(1):3–12, 1993. 153

281. G. Butts and K. Linton. The joint confidence level paradox: A history of denial. In *NASA 2009 Cost Estimating Symposium*. NASA Center for Aerospace Information, Apr. 2009. 122

282. B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1995. 176

283. E. G. Cale, L. L. Gremillion, and J. L. McKenney. Price/performance patterns of U.S. computer systems. *Communications of the ACM*, 22(4):225–233, Apr. 1979. 102

284. J. Calhoun, C. Savoie, M. Randolph-Gips, and I. Bozkurt. Human reliability analysis in spaceflight applications. *Quality and Reliability Engineering International*, 29(6):869–882, Aug. 2013. 21

285. A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt. De-anonymizing programmers via code stylometry. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 255–270, Aug. 2015. 175, 187, 195

286. C. F. Camerer and E. F. Johnson. The process-performance paradox in expert judgment: How can the experts know so much and predict so badly? In K. A. Ericsson and J. Smith, editors, *Towards a general theory of expertise: Prospects and limits*. Cambridge University Press, 1991. 38

287. J. I. D. Campbell. On the relation between skilled performance of simple division and multiplication. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 23(5):1140–1159, 1997. 48

288. M. Campbell-Kelly. *Foundations of Computer Programming in Britain (1945 - 1955)*. PhD thesis, Department of Mathematics and Computer Studies, Sunderland Polytechnic, June 1980. 102

289. M. Campbell-Kelly. *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*. The MIT Press, Apr. 2004. 58

290. M. Campbell-Kelly and D. D. Garcia-Swartz. Economic perspectives on the history of the computer time-sharing industry, 1965–1985. *IEEE Annals of the History of Computing*, 30(1):16–36, Jan.-Mar. 2008. 101

291. M. Campbell-Kelly and D. D. Garcia-Swartz. Pragmatism not ideology: IBM's love affair with open source software. Working Paper n. 1081613, UK universities, Jan. 2008. 64

292. M. Caneill and S. Zacchiroli. Debsources: Live and historical views on macro-level software evolution. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM'14, pages 28:1–28:10, Sept. 2014. 112

293. G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta. Social interactions around cross-system bug fixings: the case of FreeBSD and OpenBSD. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR'11, pages 143–152, May 2011. 245

294. L. F. Capretz, P. Waychal, and J. Jia. Comparing popularity of testing careers among Canadian, Chinese, Indian students. In *IEEE/ACM 41st International Conference on Software Engineering*, ICSE-SEET, pages 258–259, May 2019. 104

295. B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the 6^{th} Working Conference on Reverse Engineering*, WCRE'99, pages 112–122, Oct. 1999. 191

296. J. R. Carlberg. Scientific/engineering work stations: A market survey. Departmental Report DTNSRDC/CMLD-83/07, David W. Taylor Naval Ship Research and Development Center, May 1983. 111

297. S. Carter-Thomas and E. Rowley-Jolivet. *If*-conditionals in medical discourse: From theory to disciplinary practice. *Journal of English for Academic Purposes*, 7(3):191–205, July 2008. 43

298. E. Caspi. Empirical study of opportunities for bit-level specialization in word-based programs. Thesis (m.s.), University of California, Berkeley, 2000. 198

299. D. Castelvecchi. The biggest mystery in mathematics: Shinichi Mochizuki and the impenetrable proof. *Nature*, 526(7572):178–181, Oct. 2015. 144

300. M. P. Catherwood. Manpower impacts of electronic data processing. Publication B-171, New York State Department of Labor, Division of Research and Statistics, Sept. 1968. 88

301. J. P. Cavanagh. Relation between the immediate memory span and the memory search rate. *Psychological Review*, 79(6):525–530, Nov. 1972. 29

302. M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, 19(4):1040–1074, 2014. 191

303. C. Cecot and W. K. Viscusi. Judicial review of agency benefit-cost analysis. *George Mason Law Review*, 22(3):575–617, Nov. 2015. 145

304. C. Cederström and P. Fleming. *Dead Man Working*. Zero books, 2012. 63

305. A. Celik, K. Palmskog, M. Parovic, E. J. G. Arias, and M. Gligoric. Mutation analysis for Coq. In *34th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2019, pages 539–551, Nov. 2019. 145

306. D. Centola and A. Baronchelli. The spontaneous emergence of conventions: An experimental study of cultural evolution. *PNAS*, 112(7):1989–1994, Feb. 2015. 72

307. D. Centola, J. Becker, D. Brackbill, and A. Baronchelli. Experimental evidence for tipping points in social convention. *Science*, 360(6393):1116–1119, June 2018. 71

308. P. E. Ceruzzi. The early computers of Konrad Zuse, 1935 to 1945. *Annals of the History of Computing*, 3(3):241–262, July 1981. 1

309. H. S. Cha. *Disrupting the Management Supply Chain: An Organizational Learning Model of IT Offshore Outsourcing*. PhD thesis, Faculty of the Committee on Business Administration, The University of Arizona, July 2007. 73

310. F. Chandler, I. A. Heard, M. Presley, A. Burg, E. Midden, and P. Mongan. NASA human error analysis. Technical report, NASA Office of Safety and Mission Assurance, Sept. 2010. 21

311. F. T. Chandler, Y. H. J. Chang, A. Mosleh, J. L. Marble, R. L. Boring, and D. I. Gertman. Human reliability analysis methods: Selection guidance for NASA. Nasa/osma technical report, NASA Headquarters Office of Safety and Mission Assurance, July 2006. 21

312. D. Chandlera and A. Kapelner. Breaking monotony with meaning: Motivation in crowdsourcing markets. In *eprint arXiv:stat.OT/1210.0962*, Oct. 2012. 70

313. V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection–A survey. *ACM Computing Surveys*, 41(3):1–58, July 2009. 375, 376

314. K. Chandrasekar. *High-Level Power Estimation and Optimization of DRAMs*. PhD thesis, Technische Universiteit Delft, Oct. 2014. 367

315. A. C. Chang and P. Li. Is economics research replicable? Sixty published papers from thirteen journals say "usually not". Finance and Economics Discussion Series 2015-083, Washington: Board of Governors of the Federal Reserve System, Sept. 2015. 11

316. P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software–Practice and Experience*, 22(5):349–369, May 1992. 181

317. W. Chang. *R Graphics Cookbook*. O'Reilly, 2012. 221

318. A. Chao. Estimating population size for sparse data in capture–recapture experiments. *Biometrics*, 45(2):427–438, June 1989. 99

319. A. Chao, C.-H. Chiu, and L. Jost. Unifying species diversity, phylogenetic diversity, functional diversity, and related similarity and differentiation measures through Hill numbers. *Annual Review of Ecology, Evolution and Systematics*, 45(1):297–324, Nov. 2014. 92

320. A. Chao, R. K. Colwell, C.-W. Lin, and N. J. Gotelli. Sufficient sampling for asymptotic minimum species richness estimators. *Ecology*, 90(4):1125–1133, Apr. 2009. 100

321. A. Chao, S.-M. Lee, and S.-L. Jeng. Estimating population size for capture–recapture data when capture probabilities vary by time and individual animal. *Biometrics*, 48(1):201–216, Mar. 1992. 100

322. A. Chao and C.-W. Lin. Nonparametric lower bounds for species richness and shared species richness under sampling without replacement. *Biometrics*, 68(3):912–921, Sept. 2012. 99

323. A. Chao and M. C. K. Yang. Stopping rules and estimation for recapture debugging with unequal failure rates. *Biometrika*, 80(1):193–201, Mar. 1993. 171

324. C. Chapman, P. Wang, and K. T. Stolee. Exploring regular expression comprehension. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE'17, pages 405–416, Nov. 2017. 177

325. C. A. Chapman. Usage and refactoring studies of python regular expressions. Thesis (m.s.), Iowa State University, 2016. 168

326. M. R. Chapman. *In Search of Stupidity: Over 20 years of High-Tech Marketing Disasters*. Apress, second edition, 2006. 81, 88, 143

327. M. R. Chapman and R. J. Hujar. The softletter financial handbook 2011: Metrics and benchmarks mergers, IPOs, and venture finance compensation operations. website, Sept. 2011. https://softletter.com/wp-content/uploads/2017/02/FINHANDBOOK_A0055E.pdf. 58

328. P. Charbachi, L. Eklund, and E. Enoiu. Can pairwise testing perform comparably to manually handcrafted testing carried out by industrial engineers? In *eprint arXiv:cs.SE/1706.01636*, June 2017. 171

329. G. Charness and U. Gneezy. Strong evidence for gender differences in risk taking. *Journal of Economic Behavior & Organization*, 83(1):50–58, June 2012. 50

330. W. G. Chase and K. A. Ericsson. Skill and working memory. In G. H. Bower, editor, *The Psychology of Learning and Motivation*, pages 1–58. Academic Press, 1982. 38

331. N. Chater. *The Mind is Flat: The Illusion of Mental Depth and the Improvised Mind*. Allen Lane, Mar. 2018. 18

332. P. D. Chatzoglou and L. A. Macaulay. Requirements capture and analysis : A survey of current practice. *Requirements Engineering*, 1(2):75–87, June 1996. 131

333. M. Chekaf, N. Gauvrit, A. Guida, and F. Mathy. Compression in working memory and its relationship with fluid intelligence. *Cognitive Science*, 42(53):904–922, June 2018. 32

334. D. D. Chen and G.-J. Ahn. Security analysis of x86 processor microcode. Thesis (b.sc.), Arizona State University, Dec. 2014. 157

335. L. Chen, D. Wu, W. Ma, Y. Zhou, B. Xu, and H. Leung. How C++ templates are used for generic programming: An empirical study on 50 open source systems. *ACM Transactions on Software Engineering and Methodology*, 29(1):3, Feb. 2020. 180, 181

336. T. Chen, Y. Chen, Q. Guo, O. Temam, T. Wu, and W. Hu. Statistical performance comparisons of computers. In *18th International Symposium on High Performance Computer Architecture*, HPCA'12, pages 1–12, Feb. 2012. 253, 254, 256, 257

337. Y. Chen, A. Groce, X. Fern, C. Zhang, W.-K. Wong, E. Eide, and J. Regehr. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'13, pages 197–208, June 2013. 168, 314, 315

338. P. W. Cheng, K. J. Holyoak, R. E. Nisbett, and L. M. Oliver. Pragmatic versus syntactic approaches to training deductive reasoning. *Cognitive Psychology*, 18(3):293–328, July 1986. 38

339. A. Chesson and G. Chamberlin. Survey-based measures of software investment in the UK. Economic Trends 627, Office for National Statistics, UK, Feb. 2006. 58

340. R. N. Chesterman. Report of Queensland health payroll system commission of inquiry. Report, Queensland Government, Australia, July 2013. 116, 119

341. H. Cheung and S. Kemper. Competing complexity metrics and adults' production of complex sentences. *Applied Psycholinguistics*, 13:53–76, 1992. 183

342. R. C. Cheung. A user-oriented software reliability model. *IEEE Transactions on Software Engineering*, 6(2):118–125, 1980. 245

343. J. Y. Chiao, A. R. Bordeaux, and N. Ambady. Mental representations of social status. *Cognition*, 93(2):B49–B57, Sept. 2004. 48

344. J. J. Chilenski. An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Final Report DOT/FAA/AR-01/18, U.S. Department of Transportation, Federal Aviation Administration, Apr. 2001. 170

345. J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, Sept. 1994. 170

346. S. Chilton, J. Covey, M. Jones-Lee, G. Loomes, and H. Metcalf. Valuation of health benefits associated with reductions in air pollution. Technical report, Department for Environment, Food and Rural Affairs, May 2004. 148

347. C.-H. Chiu, Y.-T. Wang, B. A. Walther, and A. Chao. An improved nonparametric lower bound of species richness via a modified Good–Turing frequency formula. *Biometrics*, 70(3):671–682, Sept. 2014. 100

348. H. Cho. *System-Level Effects of Soft Errors*. PhD thesis, Department of Electrical Engineering, Stanford University, Aug. 2015. 154

349. N. Chomsky. *Syntactic Structures*. Walter de Gruyter & Co, 13th edition, 1975. 173

350. K. R. Christensen. Negative and affirmative sentences increase activation in different areas in the brain. *Journal of Neurolinguistics*, 22(1):1–17, Jan. 2009. 158

351. S. Christey and B. Martin. Buying into the bias: Why vulnerability statistics suck. blackhat USA 2013, July-Aug. 2013. 148

352. T. Christie. The widespread and persistent myth that it is easier to multiply and divide with Hindu-Arabic numerals than with Roman ones. blog: Tony Christie, Feb. 2017. https://thonyc.wordpress.com/2017/02/10/the-widespread-and-persistent-myth-that-it-is-easier-to-multiply-and-divide-with-hindu-arabic-numerals-than-with-roman-ones. 103

353. R. A. Chubon and M. R. Hester. An enhanced standard computer keyboard system for single-finger and typing-stick typing. *Journal of Rehabilitation Research and Development*, 25(4):17–24, Oct.-Dec. 1988. 92

354. A. CIA. Analytic thinking and presentation for intelligence producers: Analysis training handbook. Technical report, Office of Training and Education, Central Intelligence Agency, Aug. 199? 220

355. Z. J. Ciechanowicz and A. C. De Weever. The 'completeness' of the Pascal test suite. *Software–Practice and Experience*, 14(5):463–471, 1984. 158

356. J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall. An empirical analysis of the Docker container ecosystem on GitHub. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR'17, pages 323–333, May 2017. 110, 136

357. D. Citron. MisSPECulation: Partial and misleading use of SPEC CPU2000 in computer architecture conferences. In *Proceedings of the 30th annual International Symposium on Computer Architecture*, ISCA'03, pages 52–61, June 2003. 362

358. D. Citron and D. G. Feitelson. "look it up" or "do the math": An energy, area, and timing analysis of instruction resuse and memoization. Technical Report H-0196, International Business Machines Corporation, Oct. 2003. 359, 360, 361

359. I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer. On the number and nature of faults found by random testing. *Software Testing, Verification and Reliability*, 21(1):3–28, Mar. 2011. 166

360. Civil Service Department, UK. *Computers in Central Government Ten years ahead*. Her Majesty's Stationery Office, Jan. 1971. 101

361. H. H. Clark. *Understanding language*. Cambridge University Press, 1996. 174

362. H. H. Clark and D. Wilkes-Gibbs. Referring as a collaborative process. *Cognition*, 22:1–39, 1986. 72

363. A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009. 315

364. W. S. Cleveland. *The Elements of Graphing Data*. Wadsworth Advanced Book Program, 1985. 221

365. W. S. Cleveland and R. McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79(387):531–554, Sept. 1984. 221

366. J. Clune, J.-B. Mouret, and H. Lipson. The evolutionary origins of modularity. *Proceedings of the Royal Society B: Biological Sciences*, 280(1755):20122863, Jan. 2013. 179

367. A. Coad. Investigating the exponential age distribution of firms. *Economics: The Open-Access, Open-Assessment E-Journal*, 4(2010-17):1–30, Mar. 2010. 103

368. N. M. Coe. *The growth and locational dynamics of the UK computer services industry, 1981-1996*. PhD thesis, Department of Geography, University of Durham, 1996. 103

369. J. Coelho and M. T. Valente. Why modern open source projects fail. In *Proceedings of the 11th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, ESEC/FSE'17, pages 186–196, Sept. 2017. 118

370. J. Cohen. *Statistical Power Analysis for the Behavioural Sciences*. Routledge, second edition, 1988. 252, 254

371. J. Cohen. The Earth is round ($p < 0.05$). *American Psychologist*, 49(12):997–1003, 1994. 262

372. J. Cohen, S. Teleki, and E. Brown. *Best Kept Secrets of Peer Code Review*. SmartBear Software, 2012. 373

373. Z. Coker, S. Hasan, J. Overbey, M. Hafiz, and C. Kästner. Integers in C: An open invitation to security attacks? Technical Report CSSE14-01, Auburn University, Feb. 2014. 202

374. M. Cokol, I. Iossifov, R. Rodriguez-Esteban, and A. Rzhetsky. How many scientific papers should be retracted? *European Molecular Biology Organization*, 8(5):422–423, Apr. 2007. 9

375. R. E. Cole. *Managing Quality Fads: How American Business Learned to Play the Quality Game*. Oxford University Press, Feb. 1999. 145

376. E. G. Coleman. *Coding Freedom: The Ethics and Aesthetics of Hacking*. Princeton University Press, Dec. 2012. 103

377. M. Collard, A. Ruttle, B. Buchanan, and M. J. O'Brien. Population size and cultural evolution in nonindustrial food-producing societies. *PLoS ONE*, 8(9):e72628, Sept. 2013. 71

378. C. Collberg, T. Proebsting, and A. M. Warren. Repeatability and benefaction in computer systems research-A study and a modest proposal. Technical Report TR 14-014, Department of Computer Science, University of Arizona, Feb. 2015. 10

379. D. Comin and B. Hobijn. Cross-country technology adoption: making the theories face the facts. *Journal of Monetary Economics*, 51(1):39–83, 2004. 5

380. C. Commeyne, A. Abran, and R. Djouab. Effort estimation with story points and cosmic function points - an industry case study. *Software Measurement News*, 21(1):25–36, 2016. 125, 126

381. Committee of Public Accounts. HM revenue and customers: ASPIRE–re-competition of outsourced IT services. Technical Report Twenty-eighth Report of Session 2006-07, UK Parliament, June 2007. 96, 120

382. Comptroller General of the United States. Multiyear leasing and government-wide purchasing of automatic data processing equipment should result in significant savings. Technical Report B-115369, U.S. General Accounting Office, Apr. 1971. 101

383. Comptroller General of the United States. Federal agencies' maintenance of computer programs: Expensive and undermanaged. Technical Report AFMD-81-25, U.S. General Accounting Office, Feb. 1981. 110

384. T. Computing Technology Industry Association. Cyberstates 2019: The definitive guide to the U.S. tech industry and tech workforce. Research report, The Computing Technology Industry Association, Mar. 2019. 67

385. S. Condon, M. Regardie, M. Stark, and S. Waligora. Cost and schedule estimation study report. Technical Report SEL-93-002, Goddard Space Flight Center, Nov. 1993. 78, 129

386. Foundations for evidence-based policymaking Act of 2018 H.R.4175, Jan. 2018. 115th Congress of the United States of America, 2nd session. 2

387. M. Conoscenti, V. Besner, A. Vetrò, and D. M. Fernández. Combining data analytics and developers feedback for identifying reasons of inaccurate estimations in agile software development. *The Journal of Systems and Software*, 156:126–135, Oct. 2019. 125

388. B. Conrad and M. Mitzenmacher. Power laws for monkeys typing randomly: The case of unequal probabilities. *IEEE Transactions on Information Theory*, 50(7):1403–1414, July 2004. 243

389. J. J. Cook and C. Zilles. A characterization of instruction-level error derating and its implications for error detection. In *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC*, DSN 2008, pages 482–491, June 2008. 282

390. K. Cook. Ubuntu security hardening statistics (amd64). `https://outflux.net/ubuntu/hardening/amd64`, July 2019. 98

391. P. Coombs. *IT Project Estimation: A Practical Guide to the Costing of Software*. Cambridge University Press, 2003. 117

392. T. Copeland and V. Antikarov. *Real Options A Practitioner's Guide*. Texere Publishing Limited, Apr. 2001. 62, 63

393. A. Corazza, V. Maggio, and G. Scanniello. Coherence of comments and method implementations: a dataset and an empirical investigation. *Software Quality Journal*, 26(2):751–777, June 2018. 188

394. J. Corbet, G. Kroah-Hartman, and A. McPherson. Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it? Technical report, The Linux Foundation, Dec. 2010. 117

395. J. Corbet, G. Kroah-Hartman, and A. McPherson. Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it. Technical report, The Linux Foundation, Mar. 2012. 282

396. M. Correll and M. Gleicher. Error bars considered harmful: Exploring alternate encodings for mean and error. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2142–2151, Dec. 2014. 217

397. J. W. Cortada. *The Digital Flood: The Diffusion of Information Technology Across the U.S., Europe and Asia*. Oxford University Press, Sept. 2012. 5, 98

398. M. J. Cortese and M. M. Khanna. Age of acquisition predicts naming and lexical-decision performance above and beyond 22 other predictor variables: An analysis of 2,342 words. *The Quarterly Journal of Experimental Psychology*, 60(8):1072–1082, Aug. 2007. 190

399. L. Cosmides and J. Tooby. Evolutionary psychology: A primer. Technical report, Center for Evolutionary Psychology, University of California, Santa Barbara, 1998. 18, 42

400. D. E. Costa, S. Mujahid, R. Abdalkareem, and E. Shihab. Breaking type-safety in Go: An empirical study on the usage of the `unsafe` package. *IEEE Transactions on Software Engineering*, 14(8):???, May 2020. 192

401. D. L. Costa and M. E. Kahn. Changes in the value of life, 1940-1980. Working Paper No. 9396, National Bureau of Economic Research, USA, Dec. 2002. 149

402. V. Costan and S. Devadas. Intel SGX explained. In *Cryptology ePrint Archive: Report 2016/086*, Jan. 2016. 91

403. D. Cotroneo, A. K. Iannillo, R. Natella, and R. Pietrantuono. A comprehensive study on software aging across Android versions and vendors. In *eprint arXiv:cs.SE/2005.11523*, May 2020. 354

404. D. Cotroneo, R. Pietrantuono, S. Russo, and K. Trivedi. How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation. *The Journal of Systems and Software*, 113(C):27–43, Mar. 2016. 146

405. J. D. Couger and M. A. Colter. *Maintenance Programming: Improving Productivity Through Motivation*. Prentice-Hall, Inc, 1985. 67

406. N. Cowan. The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24(1):87–185, 2001. 29

407. M. F. Cowlishaw. Decimal floating-point: Algorism for computers. In *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, pages 104–111, June 2003. 146

408. J. F. Coyle and G. D. Polsky. Acqui-hiring. *Duke Law Journal*, 63(2):281–346, Nov. 2013. 89

409. Cray Research. *M Series Site Planning Reference Manual*. Cray Research, Inc, Apr. 1983. 90

410. W. Crooymans, P. Pradhan, and S. Jansen. Exploring network modelling and strategy in the Dutch software business ecosystem. In *Proceedings of the International Conference on Software Business*, ICSOB 2015, pages 45–59, June 2015. 104

411. F. E. Croxton and R. E. Stryker. Bar charts versus circle diagrams. *Journal of the American Statistical Association*, 22(160):473–482, Dec. 1927. 220

412. J. Culver. The life cycle of a cpu. website, 2010. `http://www.cpushack.com/life-cycle-of-cpu.html`. 250, 251

413. G. Cumming and R. Maillardet. Confidence intervals and replication: Where will the next mean fall? *Psychological Methods*, 11(3):217–227, 2006. 264

414. C. R. Cummins. *The interpretation and use of numerically-quantified expressions*. PhD thesis, Research Centre for English and Applied Linguistics, University of Cambridge, Nov. 2011. 47, 48

415. P. G. Curran and K. A. Hauser. I'm paid biweekly, just not by leprechauns: Evaluating valid-but-incorrect response rates to attention check items. *Journal of Research in Personality*, 82(103849), Oct. 2019. 371

416. B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, Nov. 1988. 126

417. B. Curtis, S. B. Sheppard, and E. Kruesi. Evaluation of software life cycle data from the PAVE PAWS project. Technical Report RADC-TR-80-28, Rome Air Development Center, Griffiss Air Force Base, Mar. 1980. 129, 130, 147, 157

418. M. A. Cusumano. Factory concepts and practices in software development: An historical overview. Working Paper #3095-89 BPS, Alfred P. Sloan School of Management, Dec. 1989. 69, 137

419. M. A. Cusumano. Shifting economies: From craft production to flexible systems and software factories. Working Paper #3325-91/BPS, Alfred P. Sloan School of Management, Aug. 1991. 137

420. M. A. Cusumano, A. Gawer, and D. B. Yoffie. *The Business of Platforms: Strategy in the Age of Digital Competition, Innovation, and Power*. Harper Business, June 2019. 105

421. K. Cwalina and B. Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison–Wesley, 2006. 179

422. J. Czerwonka. On use of coverage metrics in assessing effectiveness of combinatorial test designs. In *Sixth International Conference on Software Testing, Verification and Validation, Workshops Proceedings*, ICST 2013, pages 257–266, Mar. 2013. 169

423. J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, Feb. 2006. 163

424. A. Damasio. *Self Comes to Mind: Constructing the Conscious Brain*. Vintage books, 2012. 18

425. M. Daneman and P. A. Carpenter. Individual differences in working memory and reading. *Journal of Verbal Learning and Verbal Behavior*, 19(4):450–466, Aug. 1980. 185, 186

426. M. Daneman and P. A. Carpenter. Individual differences in integrating information between and within sentences. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 9(4):561–584, 1983. 186

427. C. Danescu-Niculescu-Mizil, R. West, D. Jurafsky, J. Leskovec, and C. Potts. No country for old members: User lifecycle and linguistic change in online communities. In *Proceedings of the 22nd international conference on World Wide Web*, WWW 2013, pages 307–318, May 2013. 189

428. B. Danglot, P. Preux, B. Baudry, and M. Monperrus. Correctness attraction: A study of stability of software behavior under runtime perturbation. *Empirical Software Engineering*, 23(4):2086–2119, Aug. 2018. 154, 156

429. A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. CPU DB: Recording microprocessor history. *Communications of the ACM*, 55(4):55–63, Apr. 2012. 92, 214, 362

430. J. Darley and C. D. Batson. "from Jerusalem to Jericho": A study of situational and dispositional variables in helping behavior. *Journal of Personality and Social Psychology*, 27(1):100–108, 1973. 21

431. P. A. David. Clio and the economics of QWERTY. *The American Economic Review*, 75(2):332–337, May 1985. 92

432. P. A. David. Computer and dynamo: The modern productivity paradox in a not-too distant mirror. No. 339, Department of Economics, Stanford University, July 1989. 4

433. J. W. Davidson, J. R. Rabung, and D. B. Whalley. Relating static and dynamic machine code measurements. Technical Report CS-89-03, Department of Computer Science, University of Virginia, July 1989. 176

434. C. J. Davis. The spatial coding model of visual word identification. *Psychological Review*, 117(3):713–758, July 2010. 29, 173

435. J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee. The impact of regular expression denial of service (ReDoS) in practice: An empirical study at the ecosystem scale. In *Proceedings of the 26th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, ESEC/FSE'18, pages 246–256, Nov. 2018. 144

436. J. C. Davis, L. G. Michael, F. Servant, C. A. Coghlan, and D. Lee. Why aren't regular expressions a lingua franca? An empirical study on the re-use and portability of regular expressions. In *Proceedings of the 27th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, ESEC/FSE'19, pages 443–454, Aug. 2019. 168

437. J. C. Davis, D. Moyer, A. M. Kazerouni, and D. Lee. Testing regex generalizability and its implications A large-scale many-language measurement study. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE'19, pages 427–439, Nov. 2019. 265, 266

438. S. J. Davis and B. S. de la Parra. Application flows. Working paper, University of Chicago Booth School of Business, Mar. 2017. 104, 110

439. S. J. Davis, J. MacCrisken, and K. M. Murphy. Economic perspectives on software design: PC operating systems and platforms. Working Paper No. 8411, National Bureau of Economic Research, USA, Aug. 2001. 1

440. S. Dayal. Characterizing HEC storage systems at rest. Technical Report CMU-PDL-08-109, Parallel Data Laboratory, Carnegie Mellon University, July 2008. 242

441. R. de Bliek. *Empirical studies on the economic impact of trust*. PhD thesis, Erasmus Research Institute of Management, Rotterdam, May 2015. 68

442. S. De Deyne, S. Verheyen, E. Ameel, W. Vanpaemel, M. J. Dry, W. Voorspoels, and G. Storms. Exemplar by feature applicability matrices and other Dutch normative data for semantic concepts. *Behavior Research Methods*, 40(4):1030–1048, Nov. 2008. 40

443. A. D. de Groot. *Thought and Choice in Chess*. Amsterdam University Press, 2008. 37

444. J. L. de la Vara, M. Borg, K. Wnuk, and L. Moonen. An industrial survey of safety evidence change impact analysis practice. *IEEE Transactions on Software Engineering*, 42(12):1095–1117, Dec. 2016. 108, 140

445. B. B. de Mesquita, A. Smith, R. M. Siverson, and J. D. Morrow. *The Logic of Political Survival*. The MIT Press, 2005. 126

446. R. A. De Millo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979. 144, 262

447. A. B. de Oliveira, J.-C. Petkovich, T. Reidemeister, and S. Fischmeister. DataMill: Rigorous performance evaluation made easy. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE'13, pages 137–148, Apr. 2013. 370, 371

448. F. G. de Oliveira Neto, R. Torkar, R. Feldt, L. Gren, C. A. Furia, and Z. Huang. Evolution of statistical analysis in empirical software engineering research: Current state and steps forward. In *eprint arXiv:cs.SE/1706.00933*, June 2017. 6

449. G. B. de Pádua and W. Shang. Revisiting exception handling practices with exception flow analysis. In *International Conference on Source Code Analysis and Manipulation*, SCAM'17, pages 11–20, Sept. 2017. 200

450. C. B. De Soto, M. London, and S. Handel. Social reasoning and spatial paralogic. *Journal of Personality and Social Psychologs*, 2(4):513–521, 1965. 44

451. K. De Vogeleer. *La loi de convexité énergie-fréquence de la consommation des programmes : modélisation, thermosensibilité et applications*. PhD thesis, Informatique [cs] Telecom ParisTech, Sept. 2015. 365

452. K. De Vogeleer, G. Memmi, and P. Jouvelot. Parameter sensitivity analysis of the energy/frequency convexity rule for nanometer-scale application processors. In *eprint arXiv:cs.DS/1508.07740*, Aug. 2015. 364

453. I. De Voldere, J.-F. Romainville, S. Knotter, E. Durinck, E. Engin, A. Le Gall, P. Kern, E. Airaghi, T. Pletosu, H. Ranaivoson, and K. Hoelck. Mapping the creative value chains: A study on the economy of culture in the digital age. Final report, Directorate-General for Education and Culture Directorate D, European Commission, 2017. 81

454. G. de Wit. Firm size distributions: An overview of steady-state distributions resulting from firm dynamics models. Technical Report N200418, EIM Business and Policy Research, Jan. 2005. 103

455. I. J. Deary. *Intelligence: A Very Short Introduction*. Oxford University Press, 2001. 50

456. B. K. Debnath, M. F. Mokbel, and D. J. Lilja. Exploiting the impact of database system configuration parameters: A design of experiments approach. *IEEE Data Engineering Bulletin*, 31(1):3–10, Mar. 2008. 361

457. T. Debsources developers. Statistics | debian sources. https://sources.debian.org/stats, June 2019. 109

458. A. Decan and T. Mens. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, ???(???):???, Nov. 2019. 113

459. A. Decan, T. Mens, and M. Claes. An empirical comparison of dependency issues in OSS packaging ecosystems. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, SANER 2017, pages 2–12, Feb. 2017. 113

460. A. Decan, T. Mens, M. Claes, and P. Grosjean. On the development and distribution of R packages: An empirical analysis of the R ecosystem. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW'15, page 41, Sept. 2015. 112

461. A. Decan, T. Mens, M. Claes, and P. Grosjean. When *GitHub* meets *CRAN*: An analysis of inter-repository package dependency problems. In *23rd International Conference on Software Analysis, Evolution, and Reengineering*, SANER'16, pages 493–504, Mar. 2016. 112

462. A. Decan, T. Mens, and E. Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *15th International Conference on Mining Software Repositories*, MSR'18, pages 181–191, May 2018. 161, 162

463. L. A. DeChurch and J. R. Mesmer-Magnus. Maintaining shared mental models over long-duration exploration missions: Literature review & operational assessment. Technical Memorandum TM-2015-218590, National Aeronautics and Space Administration, Sept. 2015. 136

464. Defence technical information center. Search page for DTIC reports, July 2016. http://dsearch.dtic.mil. 6

465. S. Dehaene. Symbols and quantities in parietal cortex: elements of a mathematical theory of number representation and manipulation. In P. Haggard, Y. Rossetti, and M. Kawato, editors, *Sensorimotor Foundations of Higher Cognition (Attention and Performance) XXII*, chapter 24, pages 527–574. Oxford University Press, Nov. 2007. 46

466. S. Dehaene. *Reading in the Brain: The Science and evolution of a human invention*. Viking, 2009. 17

467. S. Dehaene. *The Number Sense*. Oxford University Press, revised and updated edition, 2011. 44, 46

468. S. Dehaene, S. Bossini, and P. Giraux. The mental representation of parity and number magnitude. *Journal of Experimental Psychology: General*, 122(3):371–396, Sept. 1993. 20

469. S. Dehaene, E. Dupoux, and J. Mehler. Is numerical comparison digits? Analogical and symbolic effects in two-digit number comparisons. *Journal of Experimental Psychology: Human Perception and Performance*, 16(3):626–641, 1990. 47

470. S. Dehaene, V. Izard, E. Spelke, and P. Pica. Log or linear? Distinct intuitions of the number scale in Western and Amazonian indigene cultures. *Science*, 320(5880):1217–1220, May 2008. 19, 46

471. S. M. Dekleva. The influence of the information systems development approach on maintenance. *MIS Quarterly*, 16(3):355–372, Sept. 1992. 139

472. R. T. DeLamarter. *Big Blue: IBM's Use and Abuse of Power*. Pan Books, 1988. 73, 101, 126, 127

473. S. DellaVigna. Psychology and economics: Evidence from the field. Working Paper No. 13420, National Bureau of Economic Research, USA, Sept. 2007. 54

474. J. Demmel and Y. Hilda. Accurate floating point summation. Technical Report UCB//CSD-02-1180, University of California, Berkeley, May 2002. 143

475. Department of Defense. Military standard DOD-STD-2167 defense system software development. Standard DOD-STD-2167, U.S. Department of Defense, 1985. 127

476. Department of Defense. Standard practice system safety. Standard MIL-STD-882E, U.S. Department of Defense, May 2012. 149

477. Amount of end-user usage of code in Firefox. blog, July 2013. http://shape-of-code.coding-guidelines.com/2013/07/26/amount-of-end-user-usage-of-code-in-firefox. 158

478. M. Derex, J.-F. Bonnefon, R. Boyd, and A. Mesoudi. Causal understanding is not necessary for the improvement of culturally evolving technology. *Nature Human Behaviour*, 3(5):446–452, May 2019. 71

479. G. Destefanis. Which programming language should a company use? A Twitter-based analysis. Technical Report CRIM-14/10-23-MODL, Computer Research Institute of Montréal, Oct. 2014. 110

480. S. Deutsch and M. H. Jørgensen. Studying the hidden costs of offshoring – the effect of psychic distance. Thesis (m.s.), Copenhagen Business School, Aug. 2014. 123

481. J. P. DeVale. *High Performance Robust Computer Systems*. PhD thesis, Electrical and Computer Engineering, Pittsburgh, Oct. 2001. 152

482. T. Dey and A. Mockus. Deriving a usage-independent software quality metric. In *eprint arXiv:cs.SE/2002.09989*, Feb. 2020. 152

483. A. Di Franco, H. Guo, and C. Rubio-González. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE'17, pages 509–519, Nov. 2017. 157

484. C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer. Lessons learned from the analysis of system failures at petascale: The case of Blue Waters. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN 2014, pages 610–621, June 2014. 162

485. M. Di Penta, L. Cerulo, and L. Aversano. The life and death of statically detected vulnerabilities: an empirical study. *Information and Software Technology*, 51(10):1469–1484, Oct. 2009. 149, 150, 339, 340

486. A. Di Sorbo, J. Spillner, G. Canfora, and S. Panichella. "won't we fix this issue?" Qualitative characterization and automated identification of wontfix issues on GitHub. In *eprint arXiv:cs.SE/1904.02414*, Apr. 2019. 12, 145

487. T. F. Dickey. Programmer variability. *Proceedings of the IEEE*, 69(7):844–845, July 1981. 8

488. L. S. Dickstein. The effect of figure on syllogistic reasoning. *Memory & Cognition*, 6(1):76–83, 1978. 43

489. A. Diekmann. Not the first digit! Using Benford's law to detect fraudulent scientific data. *Journal of Applied Statistics*, 34(3):321–329, Oct. 2007. 382

490. J. Dietrich, K. Jezek, and P. Brada. What Java developers know about compatibility, and why this matters. In *eprint arXiv:cs.SE/1408.2607v1*, Aug. 2014. 372

491. S. Dietrich, I. Hertrich, and H. Ackermann. Training of ultra-fast speech comprehension induces functional reorganization of the central-visual system in late-blind humans. *Frontiers in Human Neuroscience*, 7(701), Oct. 2013. 17

492. E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, Mar. 1968. 199

493. C. DiMarco, G. Hirst, and M. Stede. The semantic and stylistic differentiation of synonyms and near-synonyms. In *AAAI Spring Symposium on Building Lexicons for Machine Translation*, pages 114–121, Mar. 1993. 232

494. A. Dinaburg. Bitsquatting: DNS hijacking without exploitation. Reference 2011-307, Raytheon Company, July 2011. 163

495. D. K. Dirlam. Most efficient chunk sizes. *Cognitive Psychology*, 3(2):355–359, Apr. 1972. 32

496. A. K. Dixit and R. S. Pindyck. *Investment under Uncertainty*. Princeton University Press, 1994. 61, 62, 331

497. H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2008, pages 71–82, Nov. 2008. 171

498. C. Domas. Breaking the x86 ISA. blackhat USA 2017, July 2017. 157

499. D. J. Dooling and R. E. Christiaansen. Episodic and semantic aspects of memory for prose. *Journal of Experimental Psychology: Human Learning and Memory*, 3(4):428–436, 1977. 185

500. J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'99, pages 59–70, July 1999. 242

501. J. Downer. Watching the watchmaker: On regulating the social in lieu of the technical. Discussion Paper 54, London School of Economics and Political Science, June 2009. 145

502. J. R. Doyle. Survey of time preference, delay discounting models. *Judgment and Decision Making*, 8(2):116–135, Mar. 2013. 54

503. G. Dréan. *The Computer Industry: Structure, economics, perspectives*. Gérard Dréan, english edition, 2012. 90

504. S. Drobisz, T. Mens, and R. Di Cosmo. A historical analysis of Debian package conflicts. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR'15, pages 212–223, June 2015. 113

505. S. Duffy, J. Huttenlocher, L. V. Hedges, and L. E. Crawford. Category effects on stimulus estimation: Shifting and skewed frequency distributions. *Psychonomic Bulletin & Review*, 17(2):224–230, Apr. 2010. 46

506. J. Duggan. Implementing a metapopulation Bass diffusion model using the R package deSolve. *The R Journal*, 9(1):153–163, June 2017. 352

507. R. I. M. Dunbar and R. Sosis. Optimising human community sizes. *Evolution and Human Behavior*, 39(1):106–111, Jan. 2018. 95, 96

508. J. R. Dunham and L. A. Lauterbach. An experiment in software reliability additional analyses using data from automated replications. NASA Contractor Report 178395, Research Triangle Institute, North Carolina, Jan. 1988. 153

509. J. R. Dunham and J. L. Pierce. An experiment in software reliability. NASA Contractor Report 172553, NASA Langley Research Center, Mar. 1986. 153, 154, 225

510. L. M. Dunn. *An Investigation of the Factors Affecting the Lifecycle Costs of COTS-Based Systems*. PhD thesis, School of Computing, University of Portsmouth, June 2011. 107

511. D. Dunning, C. Heath, and J. M. Suls. Flawed self-assessment: Implications for health, education, and the workplace. *Psychological Science in the Public Interest*, 5(3):69–106, Apr. 2004. 371

512. V. H. S. Durelli, J. Offutt, N. Li, M. E. Delamaro, J. Guo, Z. Shi, and X. Ai. What to expect of predicates: An empirical analysis of predicates in real world programs. *The Journal of Systems and Software*, 113:324–336, Mar. 2016. 200

513. C. Dutang. CRAN task view: Probability distributions. website, June 2016. http://CRAN.R-project.org/view=Distributions. 232, 238

514. G. Dutilh, J. Annis, S. D. Brown, P. Cassey, N. J. Evans, R. P. P. P. Grasman, G. E. Hawkins, A. Heathcote, W. R. Holmes, A.-M. Krypotos, C. N. Kupitz, F. P. Leite, V. Lerche, Y.-S. Lin, G. D. Logan, T. J. Palmeri, J. J. Starns, J. S. Trueblood, L. van Maanen, D. van Ravenzwaaij, J. Vandekerckhove, I. Visser, A. Voss, C. N. White, T. V. Wiecki, J. Rieskamp, and C. Donkin. The quality of response time data inference: A blinded, collaborative assessment of the validity of cognitive models. *Psychonomic Bulletin & Review*, 26(4):1051–1069, Aug. 2019. 20

515. T. Dybå, V. B. Kampenes, and D. I. K. Sjøberg. A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, 48(8):745–755, Aug. 2006. 6, 354

516. R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining billions of AST nodes to study actual and potential usage of Java language features. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 779–790, May-June 2014. 198

517. P. C. Earley. Social loafing and collectivism: A comparison of the United States and the People's Republic of China. *Administrative Science Quarterly*, 34(4):565–581, Dec. 1989. 68

518. H. Ebbinghaus. *Über das Gedächtnis. Untersuchungen zur experimentellen Psychologie*. Teachers College, Columbia University, 1885. Translated by Henry A. Ruger and Clara E. Bussenius as Memory: A Contribution to Experimental Psychology (Teachers College, Columbia University, 1913). 33

519. A. Eckbreth, C. Saff, K. Connolly, N. Crawford, C. Eick, M. Goorsky, N. Kacena, D. Miller, R. Schafrik, D. Schmidt, D. Stein, M. Stroscio, G. Washington, and J. Zolper. Sustaining Air Force aging aircraft into the 21st century. Technical Report SAB-TR-11-01, United States Air Force Scientific Advisory Board, Aug. 2011. 94

520. Economist Data team. The changing US technology sector: Daily chart for april 21 2015. The Economist website, Apr. 2015. As of Q1 2015, Sources: Thomson Reuters; awk scripts+R converted the data embedded in Javascript. 3

521. EDB. Offensive security's exploit database archive. https://www.exploit-db.com, Mar. 2018. 147

522. S. Eder, M. Junker, E. Jürgens, B. Hauptmann, R. Vaas, and K.-H. Prommer. How much does unused code matter for maintenance? In *34th International Conference on Software Engineering*, ICSE'12, pages 1102–1111, June 2012. 61

523. A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner. An empirical study on the effectiveness of security code review. In *Proceedings of the 5th International Conference on Engineering Secure Software and Systems*, ESSoS'13, pages 197–212, Feb. 2013. 264, 288, 297

524. M. A. Edwards and S. Roy. Academic research in the 21st century: Maintaining scientific integrity in a climate of perverse incentives and hypercompetition. *Environmental Engineering Science*, 34(1):51–61, Jan. 2017. 8

525. K. Ehrlich and P. N. Johnson-Laird. Spatial descriptions and referential continuity. *Journal of Verbal Learning and Verbal Behavior*, 21(3):296–306, June 1982. 186

526. S. G. Eick, C. R. Loader, M. D. Long, L. G. Votta, and S. V. Wiel. Estimating software fault content before coding. In *Proceedings of the 14th international conference on Software engineering*, ICSE'92, pages 59–65, May 1992. 165

527. P. Ein-Dor. Grosch's law re-revisited: CPU power and the cost of computation. *Communications of the ACM*, 28(2):142–151, Feb. 1985. 90

528. T. Eisensee and D. Strömberg. News droughts, news floods, and U.S. disaster relief. *The Quarterly Journal of Economics*, 122(2):693–728, May 2007. 148

529. K. El Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis, and S. N. Rai. The optimal class size for object-oriented software. *IEEE Transactions on Software Engineering*, 28(5):494–509, Mar. 2002. 225

530. K. El Emam and A. G. Koru. A replicated survey of IT software project failures. *IEEE Software*, 25(5):84–90, Apr. 2008. 118

531. A. Elci. The dependence of operating system size upon allocatable resources. Technical Report 75-172, Department of Computer Science, Purdue University, Dec. 1975. 106

532. I. R. Elliott. Life cycle planning for a large mix of commercial systems. In B. Elkins and L. Hunt, editors, *Software Phenomenology – Working Papers of the Software Life Cycle Management Workshop*, chapter 10, pages 203–215. Computer Systems Command, United States Army, Aug. 1977. 139

533. J. Elliott, M. Hoemmen, and F. Mueller. Exploiting data representation for fault tolerance. In *eprint arXiv:cs.NA/1312.2333v1*, Dec. 2013. 143

534. N. C. Ellis and R. A. Hennelly. A bilingual word-length effect: Implications for intelligence testing and the relative ease of mental calculation in Welsh and English. *British Journal of Psychology*, 71:43–51, 1980. 29, 358

535. P. D. Ellis. *The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results*. Cambridge University Press, 2010. 252

536. R. Engbert, A. Nuthmann, E. M. Richter, and R. Kliegl. SWIFT: A dynamical model of saccade generation during reading. *Psychological Review*, 112(4):777–813, Oct. 2005. 26

537. J. Engblom. Why SpecInt95 should not be used to benchmark embedded systems tools. *ACM SIGPLAN Notices*, 34(7):96–103, July 1999. 8

538. B. Enke and F. Zimmermann. Correlation neglect in belief formation. *The Review of Economic Studies*, 86(1):313–332, Jan. 2019. 36

539. J. Ensign and D. K. Akaka. Defense acquisitions: DOD has paid billions in award and incentive fees regardless of acquisition outcomes. Technical Report GAO-06-66, United States Government Accountability Office, Dec. 2005. 120

540. N. L. Ensmenger. Letting the "computer boys" take over: Technology and the politics of organizational transformation. *International Review of Social History*, 48(S11):153–180, Dec. 2003. 127

541. Y.-H. Eom and H.-H. Jo. Generalized friendship paradox in complex networks: The case of scientific collaboration. In *eprint arXiv:cs.SI/1401.1458*, Apr. 2014. 96

542. D. M. Erceg-Hurn and V. M. Mirosevich. Modern robust statistical methods. *American Psychologist*, 63(7):591–601, Oct. 2008. 249

543. K. A. Ericsson and N. Charness. Expert performance. *American Psychologist*, 49(8):725–747, Aug. 1994. 37

544. K. A. Ericsson and K. W. Harwell. Deliberate practice and proposed limits on the effects of practice on the acquisition of expert performance: Why the original definition matters and recommendations for future research. *frontiers in Psychology*, 10:2396, Oct. 2019. 37

545. K. A. Ericsson, R. T. Krampe, and C. Tesch-Römer. The role of deliberate practice in the acquisition of expert performance. *Psychological Review*, 100(3):363–406, July 1993. also University of Colorado, Technical Report #91-06. 38

546. K. A. Ericsson and A. C. Lehmann. Expert and exceptional performance: Evidence of maximal adaption to task constraints. *Annual Review of Psychology*, 47:273–305, 1996. 38

547. K. Eriksson, D. H. Bailey, and D. C. Geary. The grammar of approximating number pairs. *Memory & Cognition*, 38(3):333–343, Apr. 2010. 47

548. K. Eriksson, F. Jansson, and J. Sjöstrand. Bentley's conjecture on popularity toplist turnover under random copying. *The Ramanujan Journal*, 23(1-3):371–396, Dec. 2010. 72

549. N. A. Ernst, J. C. Carver, D. Mendez, and M. Torchiano. Understanding peer review of software engineering papers. In *eprint arXiv:cs.SE/2009.01209*, Sept. 2020. 9

550. L. Eshkevari, F. D. Santos, J. R. Cordy, and G. Antoniol. Are PHP applications ready for Hack? In *IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering*, SANER 2015, pages 63–72, Mar. 2015. 203

551. L. M. Eshkevari, V. Arnaoudova, M. Di Penta, R. Oliveto, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of identifier renamings. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR'11, pages 33–42, May 2011. 135

552. H. Esmaeilzadeh, T. Cao, X. Yang, S. M. Blackburn, and K. S. McKinley. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *Proceedings of the sixteenth international conference on Architectural support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 319–332, Mar. 2011. 258

553. W. K. Estes. *Classification and Cognition*. Oxford University Press, 1994. 40

554. J. A. Etzel, J. M. Zacks, and T. S. Braver. Searchlight analysis: promise, pitfalls, and potential. *NeuroImage*, 78:261–269, Sept. 2013. 173

555. A. N. Evans, B. Campbell, and M. L. Soffa. Is Rust used safely by software developers? In *Proceedings of the 42nd International Conference on Software Engineering*, ICSE'20, pages 246–257, July 2020. 192

556. J. S. B. T. Evans, J. L. Barston, and P. Pollard. On the conflict between logic and belief in syllogistic reasoning. *Memory & Cognition*, 11(3):295–306, 1983. 43

557. J. L. Eveleens and C. Verhoef. The rise and fall of the Chaos report figures. *IEEE Software*, 27(1):30–36, Jan. 2010. 118

558. J. Eyolfson, L. Tan, and P. Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR'11, pages 153–162, May 2011. 117, 323, 341

559. J. Eyolfson, L. Tan, and P. Lam. Correlations between bugginess and time-based commit characteristics. *Empirical Software Engineering*, 19(4):1009–1039, Aug. 2014. 342, 343, 344

560. Facebook. Facebook Inc. 2013 Form 10-K. website, 2014. https://www.sec.gov/Archives/edgar/data/1326801/000132680114000007/fb-12312013x10k.htm. 84

561. Facebook. Facebook Inc. 2015 Form 10-K. website, 2016. https://www.sec.gov/Archives/edgar/data/1326801/000132680116000043/fb-12312015x10k.htm. 84

562. R. Falk and C. Konold. Making sense of randomness: Implicit encoding as a basis for judgment. *Psychological Review*, 104(2):301–318, Apr. 1997. 49

563. D. Fanelli. How many scientists fabricate and falsify research? A systematic review and meta-analysis of survey data. *PLoS ONE*, 4(5):e5738, May 2009. 9

564. D. Fanelli. "Positive" results increase down the hierarchy of the sciences. *PLoS ONE*, 5(4):e10068, Apr. 2010. 10

565. F. C. Fang, R. G. Steen, and A. Casadevall. Misconduct accounts for the majority of retracted scientific papers. *PNAS*, 109(42):17028–17033, Oct. 2012. 9

566. M. Fang and M. Hafiz. Discovering buffer overflow vulnerabilities in the wild: An empirical study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM'14, pages 23:1–23:10, Sept. 2014. 148

567. L. Farr and B. Nanus. Factors that affect the cost of computer programming, volume I. Technical Documentary Report ESD-TDR-64-448, United States Air Force, L. G. Hanscom Field, Bedford, Massachusetts, July 1964. 123

568. L. Farr and H. J. Zagorski. Factors that affect the cost of computer programming, volume II: A quantitative analysis. Technical Documentary Report ESD-TDR-64-448, United States Air Force, L. G. Hanscom Field, Bedford, Massachusetts, Sept. 1964. 123

569. J. Farrell and P. Klemperer. Coordination and lock-in: Competition with switching costs and network effects. In M. Armstrong and R. H. Porter, editors, *Handbook of Industrial Organization, Volume 3*, chapter 31, pages 1967–2072. North-Holland, Oct. 2007. 96

570. J. Farrell and C. Shapiro. Dynamic competition with switching costs. *RAND Journal of Economics*, 19(1):123–137, 1988. 96

571. S. Farrell, M. J. Hurlstone, and S. Lewandowsky. Sequential dependencies in recall of sequences: Filling in the blanks. *Memory & Cognition*, 41(6):938–52, Aug. 2013. 32

572. S. Farrell, K. Oberauer, M. Greaves, K. Pasiecznik, S. Lewandowsky, and C. Jarrold. A test of interference versus decay in working memory: Varying distraction within lists in a complex span task. *Journal of Memory and Language*, 90:66–87, Oct. 2016. 31

573. FDA. General principles of software validation. Final guidance for industry and fda staff, U.S. Food and Drug Administration, Jan. 2002. 149

574. Federal Food and Drug Administration. Pma approvals. Medical device approval information, July 2019. https://www.fda.gov/medical-devices/device-approvals-denials-and-clearances/pma-approvals. 150

575. Federal Register. *United States v. Adobe Systems, Inc., et al.; Proposed Final Judgment and Competitive Impact Statement*, 2010. 75 (No. 190; October 1), 24624. 105

576. Federal Trade Commission. Dell computer corporation consent order, etc., in regard to alleged violation of sec. 5 of the federal trade commission act, docket c-3658. In P. C. Epperson, editor, *Federal Trade Commission decisions: Findings, opinions and orders volume 121*, pages 616–643. U.S. Government Printing Office, May 1996. 77

577. D. G. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press, 2014. 108, 362, 382

578. D. G. Feitelson and B. Nitzberg. Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, chapter 19, pages 337–360. Springer-Verlag, June 1995. 373

579. S. L. Feld. Why your friends have more friends than you do. *The American Journal of Sociology*, 96(6):1464–1477, May 1991. 96

580. J. Feldman. Minimization of boolean complexity in human concept learning. *Nature*, 407:630–633, Oct. 2000. 40

581. J. Feldman. An algebra of human concept learning. *Journal of Mathematical Psychology*, 50(4):339–368, Aug. 2006. 40

582. A. Feldstein and P. Turner. Overflow, underflow, and severe loss of significance in floating-point addition and subtraction. *IMA Journal of Numerical Analysis*, 6(2):241–251, Apr. 1986. 152

583. M. Felici. *Observational Models of Requirements Evolution*. PhD thesis, School of Informatics, University of Edinburgh, 2004. 140

584. S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS'10, pages 385–396, Mar. 2010. 162

585. N. Fenton, M. Neil, W. Marsh, P. Hearty, Ł. Radliński, and P. Krause. On the effectiveness of early life cycle defect prediction with Bayesian nets. *Empirical Software Engineering*, 13(5):499–537, Oct. 2008. 289, 290

586. D. V. Ferens and D. S. Christensen. Calibrating software cost models to Department of Defense databases-A review of ten studies. *ISPA Journal of Parametrics*, XVIII(2):55–74, Nov. 1998. 124

587. C. J. Ferguson and M. Heene. A vast graveyard of undead theories: Publication bias and psychological science's aversion to the Null. *Perspectives on Psychological Science*, 7(6):555–561, Nov. 2012. 11, 261

588. P. Fernández. Valuing real options: Frequently made errors. Working Paper n. 274855, Instituto de Estudios Superiores de la Empresa, Madrid, June 2001. 63

589. L. Ferrand, M. Brysbaert, E. Keuleers, B. New, P. Bonin, A. Méot, M. Augustinova, and C. Pallier. Comparing word processing times in naming, lexical decision, and progressive demasking: evidence from Chronolex. *frontiers in Psychology*, 2(306), Nov. 2011. 190

590. S. Ferson, J. O'Rawe, A. Antonenko, J. Siegrist, J. Mickley, C. C. Luhmann, K. Sentz, and A. M. Finkel. Natural language of uncertainty: numeric hedge words. *International Journal of Approximate Reasoning*, 57:19–39, Feb. 2015. 48

591. R. G. Fichman and C. F. Kemerer. Incentive compatibility and systematic software reuse. *Journal of Systems and Software*, 57(1):45–60, Apr. 2001. 77

592. A. Filippin and P. Crosetto. A reconsideration of gender differences in risk attitudes. IZA DP No. 8184, The Institute for the Study of Labor, Bonn, May 2014. 50

593. C. J. Fillmore. Topics in lexical semantics. In R. W. Cole, editor, *Current Issues in Linguistic Theory*, pages 76–138. Indiana University Press, 1977. 41

594. Financial Accounting Standards Board. Statement of financial accounting standards no. 86. Technical report, Financial Accounting Foundation, Aug. 1985. 80

595. M. Finifter. Towards evidence-based assessment of factors contributing to the introduction and detection of software vulnerabilities. Technical Report UCB/EECS-2013-49, Electrical Engineering and Computer Sciences, University of California at Berkeley, May 2013. 165

596. E. Fischer. The evolution of character codes, 1874-1968. Nov. 2002. 102

597. D. A. Fisher. A common programming language for the Department of Defense – background and technical requirements. PAPER P-1191, Institute for Defense Analyses, Science and Technology Division, June 1976. 109

598. J. Fisher and R. A. Hinde. The opening of milk bottles by birds. *British Birds*, 42(11):347–357, 1949. 71

599. J. C. Fisher and R. H. Pry. A simple substitution model of technological change. *Technological Forecasting & Social Change*, 3:75–88, Apr. 1971-1972. 82

600. P. Flajolet, P. Dumas, and V. Puyhaubert. Some exactly solvable models of urn process theory. In P. Chassaing, editor, *Proceedings of Fourth Colloquium on Mathematics and Computer Science Algorithms, Trees, Combinatorics and Probabilities*, pages 59–118, 2006. 98

601. K. Flamm. *Targeting the Computer*. The Brookings Institution, Washington, D.C., 1987. 6, 100

602. K. Flamm. *Creating the Computer*. The Brookings Institution, Washington, D.C., 1988. 1

603. K. Flamm. Measuring Moore's law: Evidence from price, cost, and quality indexes. Working Paper No. 24553, National Bureau of Economic Research, USA, Apr. 2018. 5

604. D. Flater. Estimation of uncertainty in application profiles. NIST TN.1826, National Institute of Standards and Technology, Apr. 2014. 370

605. D. Flater. Screening for factors affecting application performance in profiling measurements. NIST Technical Note 1855, National Institute of Standards and Technology, Oct. 2014. 368

606. D. Flater and W. F. Guthrie. A case study of performance degradation attributable to run-time bounds checks on C++ vector access. *Journal of Research of the National Institute of Standards and Technology*, 118(012):260–279, May 2013. 198, 291, 293

607. P. J. Fleming and J. J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, Mar. 1986. 363

608. J. I. Flombaum, J. A. Junge, and M. D. Hauser. Rhesus monkeys (*Macaca mulatta*) spontaneously compute addition operations over large numbers. *Cognition*, 97(3):315–325, Oct. 2005. 46

609. B. Floyd, T. Santander, and W. Weimer. Decoding the representation of code in the brain: An fMRI study of code review and expertise. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE'17, pages 175–186, May 2017. 173

610. B. Flyvbjerg. How planners deal with uncomfortable knowledge: The dubious ethics of the American Planning Association. *Cities*, 32:157–163, June 2013. 123

611. B. Flyvbjerg, M. S. Holm, and S. L. Buhl. Underestimating costs in public works projects: Error or lie? *Journal of the American Planning Association*, 68(3):279–295, June 2002. 121

612. J. Fodor. *The Modularity of Mind: An Essay on Faculty Psychology*. MIT Press, 1983. 18

613. R. A. Foley. An evolutionary and chronological framework for human social behaviour. *Proceedings of the British Academy*, 88:95–117, 1996. 17

614. P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys'17, pages 328–343, Apr. 2017. 164

615. R. E. Fontana Jr. and G. M. Decad. Moore's law realities for recording systems and memory storage components: HDD, tape, NAND, and optical. *AIP Advances*, 8(5):056506, May 2018. 94

616. C. E. Ford and S. A. Thompson. Conditionals in discourse: A text-based study from English. In E. C. Traugott, A. T. Meulen, J. S. Reilly, and C. A. Furguson, editors, *On Conditionals*, chapter 18, pages 353–372. Cambridge University Press, 1986. 43

617. C. Foroughi and A. D. Stern. Digital innovation with high costs of entry: Evidence from software-driven medical devices. HBS Working Paper #18-094, Harvard Business School, Mar. 2018. 137

618. J. Förster, E. T. Higgins, and A. T. Bianco. Speed/accuracy decisions in task performance: Built-in trade-off or separate strategic concerns? *Organizational Behavior and Human Decision Processes*, 90(1):148–164, Jan. 2003. 22

619. J. Fowkes and C. Sutton. Parameter-free probabilistic API mining across GitHub. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 254–265, Nov. 2016. 348, 349

620. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison–Wesley, 1999. 7

621. W. B. Frakes, C. J. Fox, and B. A. Nejmeh. *Software Engineering in the Unix/C Environment*. Prentice-Hall, Inc, 1991. 179

622. S. Frederick, G. Loewenstein, and T. O'Donoghue. Time discounting: A critical review. *Journal of Economic Literature*, 40(2):351–401, June 2002. 174

623. D. P. Freedman and G. M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews*. Dorset House Publishing, 1990. 164

624. P. A. Freund and N. Kasten. How smart do you think you are? A meta-analysis on the validity of self-estimates of cognitive ability. *Psychological Bulletin*, 138(2):296–321, Mar. 2011. 19

625. A. Frumusanu. The Samsung Exynos 7420 deep dive - Inside a modern 14nm SoC. website, June 2015. `http://www.anandtech.com/show/9330/exynos-7420-deep-dive/5`. 364, 365

626. W.-T. Fu and W. D. Gray. Memory versus perceptual-motor trade-offs in a blocks world task. In *Proceedings of the Twenty-second Annual Conference of the Cognitive Science Society*, pages 154–159. Erlbaum, 2000. 23

627. Y. Funami and M. H. Halstead. A software physics analysis of akiyama's debugging data. Technical Report CSD-TR 144, Purdue University, May 1975. 194

628. B. M. Fung, K. Wang, R. Chen, and P. S. Yu. Privacy-preserving data publishing: A survey of recent developments. *ACM Computing Surveys*, 42(4):1–53, June 2010. 374

629. C. A. Furia. Bayesian statistics in software engineering: Practical guide and case studies. In *eprint arXiv:cs.SE/1608.06865*, Aug. 2016. 250

630. G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. Statistical semantics: Analysis of the potential performance of keyword information systems. *The Bell System Technical Journal*, 62(6):1753–1805, July-Aug. 1983. 102

631. G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, Nov. 1987. 102

632. T. Futagami, M. Itoh, Y. Mihara, F. Mitsuhashi, H. Nishiyama, M. Shukuguchi, N. Tachi, K. Toyama, H. Obata, Y. Ooizumi, T. Shimizu, and S. Takeichi. *ESCR Embedded System development Coding Reference guide [C Language Edition]*. Information-technology Promotion Agency, Japan, 2.0 edition, 2017. 179

633. R. Futrell, K. Mahowald, and E. Gibson. Large-scale evidence of dependency length minimization in 37 languages. *PNAS*, 112(33):10336–10341, Aug. 2015. 182, 183

634. M. T. Gailliot and R. F. Baumeister. The physiology of willpower: Linking blood glucose to self-control. *Personality and Social Psychology Review*, 11(4):303–327, Nov. 2007. 55

635. W. A. Gale. Good-Turing smoothing without tears. Technical Report 94.5, AT&T Bell Laboratories, Aug. 1994. 379

636. K. Gallaba, C. Macho, M. Pinzger, and S. McIntosh. Noise and heterogeneity in historical build data: An empirical study of Travis CI. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*, ASE'18, pages 87–97, Sept. 2018. 136

637. K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *International Symposium on Empirical Software Engineering and Measurement*, ESEM'15, pages 247–256, Oct. 2015. 203, 205

638. C. R. Gallistel, S. Fairhurst, and P. Balsam. The learning curve: Implications of a quantitative analysis. *PNAS*, 101(36):13124–13131, Sept. 2004. 33

639. C. R. Gallistel, M. Krishan, Y. Liu, R. Miller, and P. E. Latham. The perception of probability. *Psychological Review*, 121(1):96–123, Jan. 2014. 49

640. T. J. Gandomani, K. T. Wei, and A. K. Binhamid. A case study research on software cost estimation using experts' estimates, Wideband Delphi, and Planning Poker technique. *International Journal of Software Engineering and Its Applications*, 8(11):173–182, Apr. 2014. 270

641. A. Gandy. *The entry of established electronics companies into the early computer industry in the UK and USA*. PhD thesis, London School of Economics and Political Science, 1992. 1, 108

642. J. D. Gannon. An experimental evaluation of data type conversions. *Communications of the ACM*, 20(8):584–595, Aug. 1977. 201

643. Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang. Making system user interactive tests repeatable: When and what should we control? In *Proceedings of the 37th International Conference on Software Engineering*, ICSE'15, pages 55–65, May 2015. 170

644. M. K. Gardner, E. Z. Rothkopf, R. Lapan, and T. Laferty. The word frequency effect in lexical decision: Finding a frequency-based component. *Memory & Cognition*, 15(1):24–28, 1987. 190

645. M. R. Garman. The generalizability of private sector research on software project management in two USAF organizations: An exploratory study. Thesis (m.s.), Air Force Institute of Technology, USA, Mar. 2003. 135

646. R. Garner and F. R. Dill. The legendary IBM 1401 data processing system. *IEEE Solid-State Circuits Magazine*, 2(1):28–39, Jan. 2010. 101

647. V. Garousi, M. Borg, and M. Oivo. Cut to the chase: Revisiting the relevance of software engineering research. In *eprint arXiv:cs.SE/1812.01395*, Dec. 2018. 6

648. Gartner. Worldwide smartphone sales. https://en.wikipedia.org/wiki/Mobile_operating_system, July 2017. 3, 89

649. J. Gascoigne. Introducing open salaries at buffer our transparent formula and all individual salaries buffer. website, Dec. 2013. https://open.buffer.com/introducing-open-salaries-at-buffer-including-our-transparent-formula-and-all-individual-salaries. 67

650. B. Gates. Shell plans - iShellBrowser. Plaintiff's Exhibit 2151, JOE COMES, RILEY PAINT, INC., SKEFFINGON'S FORMAL WEAR, INC., PATRICIA ANNE LARSEN vs. MICROSOFT CORPORATION; IOWA District Court for Polk County, Oct. 1994. 113

651. D. C. Gause and G. M. Weinberg. *Exploring Requirements: Quality before design*. Dorset House Publishing, 1989. 131

652. G. Gay, A. Rajan, M. Staats, M. Whalen, and M. P. E. Heimdahl. The effect of program and model structure on the effectiveness of MC/DC test adequacy coverage. *ACM Transactions on Software Engineering and Methodology*, 25(3):25, Aug. 2016. 170

653. J. E. Gayek, L. G. Long, K. D. Bell, R. M. Hsu, and R. K. Larson. Software cost and productivity model. Technical Report ATR-2004(8311)-1, Aerospace Corporation, Feb. 2004. 79, 80

654. Gcc releases. Vendor: website, July 2019. https://gcc.gnu.org/releases.html. 91, 114

655. Y. Ge and B. Xu. Dynamic staffing and rescheduling in software project management: A hybrid approach. *PLoS ONE*, 11(6):e0157104, June 2016. 126, 128

656. Y. Geffen and S. Maoz. On method ordering. In *IEEE 24th International Conference on Program Comprehension*, ICPC'16, pages 1–10, May 2016. 205

657. W. Gellerich, M. Kosiol, and E. Ploedereder. Where does GOTO go to? In *Reliable Software Technology – Ada-Europe 1996*, volume 1088 of *LNCS*, pages 385–395. Springer, 1996. 199

658. S. A. Gelman and E. M. Markman. Categories and induction in young children. *Cognition*, 23:183–209, 1986. 38

659. D. Gentner and S. Goldin-Meadow. *Language In Mind: Advances in the Study of Language and Thought*. MIT Press, 2003. 196

660. S. L. Gerhart and L. Yelowitz. Observations of fallibility in applications of modern programming methodologies. *IEEE Transactions on Software Engineering*, SE-2(3):195–207, Sept. 1976. 164

661. M. Gerlach, B. Farb, W. Revelle, and L. A. N. Amaral. A robust data-driven approach identifies four personality types across four large data sets. *Nature Human Behaviour*, 2(10):735–742, Sept. 2018. 50

662. D. M. German, B. Adams, and A. E. Hassan. Continuously mining distributed version control systems: An empirical study of how Linux uses git. *Empirical Software Engineering*, 21(1):260–299, Feb. 2016. 12, 374

663. D. M. German and J. M. González-Barahona. An empirical study of the reuse of software licensed under the GNU general public license. In *The 5th International Conference on Open Source Systems*, OSS 2009, pages 185–198, June 2009. 64

664. D. M. German, Y. Manabe, and K. Inoue. A sentence-matching method for automatic license identification of source code files. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE'10, pages 437–446, Apr. 2010. 65

665. E. H. Gibbs, G. A. Munroe, A. M. Zeman, and C. T. Cottingham. JANET SKOLD and DAVID DOSSANTOS, on behalf of themselves and all others similarly situated and the general public, v. INTEL CORPORATION, HEWLETT PACKARD COMPANY and DOES 1-50, case no. 1-05-CV-039231, filing #g-43414. Opinion, Superior court of the state of California for the county of Santa Clara, 2012. 362

666. E. Gibson. Linguistic complexity: locality of syntactic dependencies. *Cognition*, 68(1):1–76, Aug. 1998. 182

667. E. Gibson and J. Thomas. Memory limitations and structured forgetting: The perception of complex ungrammatical sentences as grammatical. *Language and Cognitive Processes*, 14(3):225–248, 1999. 186

668. G. Gigerenzer. Striking a blow for sanity in theories of rationality. In M. Augier and J. G. March, editors, *Models of a man: Essays in memory of Herbert A. Simon*, pages 389–409. MIT Press, May 2004. 51

669. G. Gigerenzer. *Rationality for Mortals-How People cope with Uncertainty*. Oxford University Press, 2008. 41, 262

670. G. Gigerenzer, W. Gaissmaier, E. Kurz-Milcke, L. M. Schwartz, and S. Woloshin. Helping doctors and patients make sense of health statistics. *Psychological Science in the Public Interest*, 8(2):53–96, Apr. 2008. 220

671. G. Gigerenzer, S. Krauss, and O. Vitouch. The null ritual: What you always wanted to know about significance testing but were afraid to ask. In D. Kaplan, editor, *The Sage handbook of quantitative methodology for the social sciences*, chapter 21, pages 391–408. Sage Publications, Inc, 2004. 263

672. G. Gigerenzer, P. M. Todd, and The ABC Research Group. *Simple Heuristics That Make Us Smart*. Oxford University Press, 1999. 18, 41, 51

673. B. Gilchrist and R. E. Weber. Employment of trained computer personnel–A quantitative survey. In *Proceedings of the Spring Joint Computer Conference*, AFIPS'72, pages 641–648, May 1972. 104

674. J. Gimpel. Software that checks software: The impact of PC-lint. *IEEE Software*, 31(1):15–19, Jan.-Feb. 2014. 140

675. V. Girotto, A. Mazzocco, and A. Tasso. The effect of premise order on conditional reasoning: a test of the mental model theory. *Cognition*, 63:1–28, 1997. 43

676. M. Givon, V. Mahajan, and E. Muller. Software piracy: Estimation of lost sales and the impact on software diffusion. *Journal of Marketing*, 59(1):29–37, Jan. 1995. 83, 328

677. T. J. Glauthier. Computer time sharing: Its origins and development. *Computers and Automation*, 16(10):23–27, Oct. 1967. 1

678. A. Glenberg. Few believe the world is flat: How embodiment is changing the scientific understanding of cognition. *Canadian Journal of Experimental Psychology*, 69(2):165–171, June 2015. 20

679. F. Gobet. *Understanding Expertise: A Multi-disciplinary Approach*. Palgrave, 2016. 37

680. D. R. Godden and A. D. Baddeley. Context-dependent memory in two natural environments: On land and underwater. *British Journal of Psychology*, 66(3):325–331, 1975. 31

681. M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In 16^{th} *International Conference on Software Maintenance*, ICSM'00, pages 131–142, Oct. 2000. 287

682. M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, Feb. 2005. 206

683. A. L. Goel. An experimental investigation into software reliability. Final Technical Report RADC-TR-88-213, CASE Center, Syracuse University, Oct. 1988. 159

684. M. Goeminne and T. Mens. Towards a survival analysis of database framework usage in Java projects. In *31st International Conference on Software Maintenance and Evolution*, ICSME 2015, pages 551–556, Sept.-Oct. 2015. 340

685. S. S. Gokhale and R. E. Mullen. The marginal value of increased testing: An empirical analysis using four code coverage measures. *Journal of the Brazilian Computer Society*, 12(3):13–30, Dec. 2006. 171

686. M. M. Gold. A methodology for evaluating time-shared computer system usage. Technical report, Carnegie Mellon University, Aug. 1967. 113

687. K. Goldberg, T. Roeder, D. Gupta, and C. Perkins. Eigentaste: A constant time collaborative filtering algorithm. In *Information Retrieval*, 4, pages 133–151, July 2001. 230

688. L. R. Goldberg. An alternative "description of personality": The big-five factor structure. *Journal of Personality and Social Psychologs*, 59(6):1216–12297, 1990. 50

689. L. R. Goldberg, J. A. Johnson, H. W. Eber, R. Hogan, M. C. Ashton, C. R. Cloninger, and H. G. Gough. The international personality item pool and the future of public-domain personality measures. *Journal of Research in Personality*, 40(1):84–96, 2006. 50

690. M. S. Goldberg and A. Touw. Statistical methods for learning curves and cost analysis. Technical Report CIMD0006870.A3/1Rev, The CNA Corporation, Mar. 2003. 72

691. H. H. Goldstine and J. von Neumann. Planning and coding of problems for an electronic computing instrument. Technical Report Part II, Volume 1-3, Institute for Advanced Study, Princeton, Apr. 1947. 127

692. P. Golle. Revisiting the uniqueness of simple demographics in the US population. In *Proceedings of the 5th ACM workshop on Privacy in electronic society*, WPES'06, pages 77–80, Oct. 2006. 374

693. R. W. Gomulkiewicz. Enforcement of open source software licenses: The MDY trio's inconvenient compliations. *Yale Journal of Law & Technology*, 14:106–137, 2011. 66

694. I. R. Gonzaga, Jr. Empirical studies on fine-grained feature dependencies. Thesis (m.s.), Universidade Federal de Alagoas, Instituto de Computação, Aug. 2015. 203

695. R. Gonzalez and G. Wu. On the shape of the probability weighting function. *Cognitive Psychology*, 38(1):129–166, Feb. 1999. 49

696. J. M. González-Barahona, G. Robles, I. Herraiz, and F. Ortega. Studying the laws of software evolution in a long-lived FLOSS project. *Journal of Software: Evolution and Process*, 26(7):589–612, July 2014. 216, 253, 254, 314, 329

697. B. H. Good, Y.-A. de Montjoye, and A. Clauset. The performance of modularity maximization in practical contexts. In *eprint arXiv:physics.data-an/0910.0165v2*, Apr. 2010. 245

698. J. Goodman. Lessons learned from seven Space Shuttle missions. NASA Contractor Report CR-2007-213697, Lyndon B. Johnson Space Center, Jan. 2007. 144

699. P. Goodridge, J. Haskel, and G. Wallis. Estimating UK investment in intangible assets and intellectual property rights. Technical Report No. 2014/36, Intellectual Property Office, UK government, Sept. 2014. 4

700. Google books ngram dataset. website, 2015. http://storage.googleapis.com/books/ngrams/books/datasetsv2.html. 381

701. A. Gopal and B. R. Koka. The role of contracts on quality and returns to quality in offshore software development outsourcing. *Decision Sciences*, 41(3):491–516, Aug. 2010. 120

702. R. Gopinath. *On the Limits of Mutation Analysis*. PhD thesis, Oregon State University, June 2017. 171

703. R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce. How hard does mutation analysis have to be, anyway? In *IEEE 26th International Symposium on Software Reliability Engineering*, ISSRE 2015, pages 216–227, Nov. 2015. 171, 231

704. R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 72–82, June 2014. 170, 171, 302

705. R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *IEEE 25th International Symposium on Software Reliability Engineering*, ISSRE'14, pages 189–200, Nov. 2014. 160, 171

706. R. D. Gordon and M. H. Halstead. An experiment comparing Fortran programming times with the software physics hypothesis. Technical Report TR 167, Purdue University, Oct. 1975. 194

707. R. J. Gordon. The postwar evolution of computer prices. Working Paper No. 2227, National Bureau of Economic Research, USA, Apr. 1987. 3

708. M. Gottscho. ViPZonE: Exploiting DRAM power variability for energy savings in Linux x86-64. Thesis (m.s.), Electrical Engineering, UCLA, Mar. 2014. 367

709. M. Gottscho, A. A. Kagalwalla, and P. Gupta. Power variability in contemporary DRAMs. *IEEE Embedded Systems Letters*, 4(12):37–40, June 2012. 367

710. S. Götz, T. Ilsche, J. Cardoso, J. Spillner, U. Aßmann, W. Nagel, and A. Schill. Energy-efficient data processing at sweet spot frequencies. In *OTM Workshops*, 2014, pages 154–171, Apr. 2014. 364

711. G. Gousios and A. Zaidman. A dataset for pull-based development research. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR'14, pages 368–371, May 2014. 213

712. G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE'15, pages 358–368, May 2015. 218, 219

713. K. Goševa-Popstojanova, M. Hamill, and R. Perugupalli. Large empirical case study of architecture-based software reliability. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, ISSRE'05, pages 43–52, Nov. 2005. 244

714. E. M. Grabbe, S. Ramo, and D. E. Wooldridge. *Handbook of Automation, Computation, and Control, Volume 2: Computers and Data Processing*. John Wiley & Sons, Inc, 1959. 109

715. P. Grady. *Termination of the SIREN ICT project*. Grant Thornton UK LLP, June 2014. 128

716. R. B. Grady and D. L. Caswell. *Software Metrics: Establishing a company-wide program*. Prentice-Hall, Inc, 1987. 147

717. A. C. Graesser, S. B. Woll, D. J. Kowalski, and D. A. Smith. Memory for typical and atypical actions in scripted activities. *Journal of Experimental Psychology: Human Learning and Memory*, 6(5):503–515, June 1980. 184, 185

718. S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière. Autotuning for floating-point precision with discrete stochastic arithmetic. HAL Id: hal-01331917, HAL archives-ouvertes.fr, June 2016. 146

719. E. E. Grant and H. Sackman. An exploratory investigation of programmer performance under on-line and off-line conditions. *IEEE Transactions on Human Factors in Electronics*, 8(1):33–48, Mar. 1967. 8, 55, 271

720. Graphviz-graph visualization software. website, 2015. http://www.graphviz.org. 218

721. C. A. Graver, W. M. Carriere, E. E. Balkovich, and R. Thibodeau. Cost reporting elements and activity cost tradeoffs for defense system software (study results). Technical Report ESD-TR-77-262, Vol. 1, General Research Corporation, May 1977. 129

722. D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. The misuse of the NASA metrics data program data sets for automated software defect prediction. In *15th Annual Conference on Evaluation & Assessment in Software Engineering 2011*, EASE 2011, pages 96–103, Apr. 2011. 374

723. J. Gray, C. Nyberg, M. Shah, and N. Govindaraju. Sort benchmark. http://sortbenchmark.org, July 2014. 362

724. K. Gray, D. G. Rand, E. Ert, K. Lewis, S. Hershman, and M. I. Norton. The emergence of "us and them" in 80 lines of code: Modeling group genesis in homogeneous populations. *Association for Psychological Science*, 25(4):982–990, Apr. 2014. 74

725. W. D. Gray, C. R. Sims, W.-T. Fu, and M. J. Schoelles. The soft constraints hypothesis: A rational analysis approach to resource allocation for interactive behavior. *Psychological Review*, 113(3):461–482, July 2006. 23

726. M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM'10, pages 11:1–11:10, Sept. 2010. 196

727. J. H. Greenberg. Some universals of grammar with particular reference to the order of meaningful elements. In J. H. Greenberg, editor, *Universals of Language*, chapter 5, pages 58–90. MIT Press, 1963. 43

728. H. I. Greenfield. An economist looks at data processing. *Computers and Automation*, 6(10):18–23, Oct. 1957. 101

729. S. Greenstein. Did computer technology diffuse quickly?: Best and average practice in mainframe computers, 1968-1983. Working Paper No. 4647, National Bureau of Economic Research, USA, Feb. 1994. 95

730. C. Gregg and K. Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 134–144, Apr. 2011. 357

731. P. Grice. *Studies in the Way of Words*. Harvard University Press, 1989. 173, 174

732. D. A. Grier. The ENIAC, the verb "to program" and the emergence of digital computers. *IEEE Annals of the History of Computing*, 18(I):51–55, 1996. 1

733. D. A. Grier. *When Computers were Human*. Princeton University Press, 2005. 1, 88

734. S. Grimstad and M. Jørgensen. Inconsistency of expert judgement-based estimates of software development effort. *Journal of Systems and Software*, 80(11):1770–1777, Nov. 2007. 122

735. R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL 4 Programming Language*. Prentice-Hall, Inc, second edition, 1968. 168

736. E. Grochowski and R. E. Fontana, Jr. Future technology challenges for NAND flash and HDD products. Flash Memory Summit 2012, Santa Clara, CA, July 2012. 314

737. U. Grömping. Relative importance for linear regression in R: The package `relaimpo`. *Journal of Statistical Software*, 17(1):1–27, Sept. 2006. 305

738. E. H. B. M. Gronenschild, P. Habets, H. I. L. Jacobs, R. Mengelers, N. Rozendaal, J. van Os, and M. Marcelis. The effects of FreeSurfer version, workstation type, and Macintosh operating system version on anatomical volume and cortical thickness measurements. *PLoS ONE*, 7(6):e38234, June 2012. 146

739. H. R. J. Grosch. High speed arithmetic: The digital computer as a research tool. *Journal of the Optical Society of America*, 43(4):306–310, Apr. 1953. 1

740. A. S. Grove. *Only the Paranoid Survive: How to Exploit the Crisis Points That Challenge Every Company and Career*. HarperCollinsBusines, Apr. 1988. 88

741. A. Grübler and N. Nakićenović. Long waves, technology diffusion, and substitution. Technical Report RP-91-17, International Institute for Applied Systems Analysis Laxenburg, Austria, Oct. 1991. 2

742. W. Gruhl. Lessons learned cost/schedule assessment guide. Slides of talk, July 199? 214

743. M. Gubler. *Protean and boundaryless career orientations - an empirical study of IT professionals in Europe*. PhD thesis, Loughborough University, July 2011. 67

744. T. Gue. Triggering infection: Distribution and derivative works under the GNU general public license. *Journal of Law, Technology & Policy*, 2012(1):95–140, 2012. 64

745. L. Guerrouj, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. An experimental investigation on the effects of context on source code identifiers splitting and expansion. *Empirical Software Engineering*, 19(6):1706–1753, Dec. 2014. 358

746. A. Gunasekaran, E. W. T. Ngai, and R. E. McGaughey. Information technology and systems justification: A review for research and applications. *European Journal of Operational Research*, 173(3):957–983, Sept. 2006. 115

747. H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing*, SOCC'14, pages 1–14, Nov. 2014. 275

748. H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. L. Liblit. EIO: Error handling is Occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 207–222, Feb. 2008. 159

749. N. J. Gunther. A simple capacity model of massively parallel transaction systems. In *Proceedings of 19th International CMG Conference*, pages 1035–1044, Dec. 1993. 223

750. N. J. Gunther. *Analysing Computer System Performance with Perl::PDQ*. Springer-Verlag, 2005. 223, 362

751. J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wąsowski. Variability-aware performance prediction: A statistical learning approach. In *IEEE/ACM 28th International Conference on Automated Software Engineering*, ASE'13, pages 301–311, Nov. 2013. 360

752. O. Gurevich, M. A. Johnson, and A. E. Goldberg. Incidental verbatim memory for language. *Language and Cognition*, 2(1):45–78, May 2010. 183

753. R. K. Guy. The strong law of small numbers. *American Mathematical Monthly*, 95(8):697–712, Oct. 1988. 251

754. E. A. E. Habib. Geometric mean for negative and zero values. *International Journal of Research & Reviews in Applied Sciences*, 11(3):419–432, June 2012. 258

755. Hackerone. The 2018 hacker report. Technical report, hackerone, Dec. 2017. 64

756. J. Haidt. *The Righteous Mind*. Vintage books, 2012. 41

757. S. Haine. As low as reasonably practicable (ALARP) risk-informed decision framework applied to public utility safety. Staff white paper, California Public Utilities Commission, Dec. 2015. 143

758. A. G. Haldane and R. Davies. The short long. Speech, May 2011. 29th Société Universitaire Européene de Recherches Financiéres Colloquium. 103

759. A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry. Test them all, is it worth it? A ground truth comparison of configuration sampling strategies. In *eprint arXiv:cs.SE/1710.07980*, Oct. 2017. 136, 164

760. T. Halkjelsvik and M. Jørgensen. *Time Predictions: Understanding and Avoiding Unrealism in Project Planning and Everyday Life*. Springer International Publishing AG, Apr. 2018. 125

761. B. H. Hall and M. MacGarvie. The private value of software patents. *Research Policy*, 39(7):994–1009, Sept. 2010. 64

762. T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, Nov. 2012. 374

763. M. H. Halstead. A theoretical relationship between mental work and machine language programming. Technical Report CSD TR 67, Purdue University, Feb. 1972. 194

764. M. H. Halstead and P. M. Zislis. Experimental verification of two theorems of software physics. Technical Report TR 97, Purdue University, June 1973. 194

765. D. Z. Hambrick, F. L. Oswald, E. M. Altmann, E. J. Meinz, F. Gobet, and G. Campitelli. Deliberate practice: Is that all it takes to become an expert? *Intelligence*, 45(1):34–45, July-Aug. 2014. 37

766. M. Hamill and K. Goseva-Popstojanova. Exploring the missing link: an empirical study of software fixes. *Software Testing, Verification and Reliability*, 24(8):684–705, Dec. 2014. 219, 220

767. M. T. Hannan and G. R. Carroll. *Dynamics of Organizational Populations: Density, Legitimation, and Competition*. Oxford University Press, Jan. 1992. 95

768. J. E. Hannay, D. I. K. Sjøberg, and T. Dybå. A systematic review of theory use in software engineering experiments. *IEEE Transactions on Software Engineering*, 33(2):87–107, Feb. 2007. 6

769. M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. Report No. UCB/CSD-95-887, Computer Science Division, University of California Berkeley, Nov. 1995. 108

770. D. Harhoff, B. H. Hall, G. von Graevenitz, K. Hoisl, S. Wagner, A. Gambardella, and P. Giuri. The strategic use of patents and its implications for enterprise and competition policies. Final Report ENTR/05/82, DG Enterprise, European Commission, July 2007. 64

771. B. R. Harmon and N. I. Om. Schedule assessment methods for ballistic missile defense ground-based software development. IDA Paper P-3600, Institute for Defense Analyses, Aug. 2003. 124

772. N. Harrand, S. Allier, M. Rodriguez-Cancio, M. Monperrus, and B. Baudry. A journey among Java neutral program variants. In *eprint arXiv:cs.SE/1901.02533*, Jan. 2019. 175

773. A. Hart, L. Maxim, M. Siegrist, N. Von Goetz, C. da Cruz, C. Merten, O. Mosbach-Schulz, M. Lahaniatis, A. Smith, and A. Hardy. Guidance on communication of uncertainty in scientific assessments. *EFSA Journal*, 17(1):5520, Jan. 2019. 226

774. T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. *ACM Transactions on Computer Systems*, 30(3):10, Aug. 2012. 369, 371

775. J. Haskel and S. Westlake. *Capitalism without Capital: The Rise of the Intangible Economy*. Princeton University Press, 2018. 4, 57

776. H. Hata, C. Treude, R. G. Kula, and T. Ishio. 9.6 million links in source code comments: Purpose, evolution, and decay. In *eprint arXiv:cs.SE/1901.07440*, Jan. 2019. 114

777. L. Hatton. *Safer C : Developing Software for High-integrity and Safety-critical Systems*. McGraw-Hill, 1995. 179

778. L. Hatton. Reexamining the fault density-component size connection. *IEEE Software*, 14(2):89–97, Mar. 1997. 225

779. L. Hatton. How accurately do engineers predict software maintenance tasks? *Computer*, 40(2):64–69, Feb. 2007. 140, 141, 216

780. M. D. Hauser, S. Carey, and L. B. Hauser. Spontaneous number representation in semi-free-ranging rhesus monkeys. *Proceedings of the Royal Society B: Biological Sciences*, 267(1445):829–833, Apr. 2000. 18

781. J. P. Haverty and R. L. Patrick. Programming languages and standardization in command and control. Research Memorandum RM-3447-PR, The RAND Corporation, Jan. 1963. 109

782. D. M. Hawkins. *Identification of Outliers*. Springer, 1980. 375

783. G. Hawkins, S. D. Brown, M. Steyvers, and E.-J. Wagenmakers. Context effects in multi-alternative decision making: Empirical data and a Bayesian model. *Cognitive Science*, 36(3):498–516, Apr. 2012. 56

784. G. E. Hawkins, S. D. Brown, M. Steyvers, and E.-J. Wagenmakers. An optimal adjustment procedure to minimize experiment time in decisions with multiple alternatives. *Psychonomic Bulletin & Review*, 19(2):339–348, Apr. 2012. 358

785. J. A. Hawkins. *Efficiency and Complexity in Grammars*. Oxford University Press, 2007. 182

786. B. Hayes. Third base. *American Scientist*, 89(6):490–494, 2001. 5

787. S. Hazelhurst. Truth in advertising: Reporting performance of computer programs, algorithms and the impact of architecture and systems environment. *South African Computer Journal*, 46:24–37, Dec. 2010. 312

788. M. L. Head, L. Holman, R. Lanfear, A. T. Kahn, and M. D. Jennions. The extent and consequences of p-hacking in science. *PLoS Biology*, 13(3):e1002106, Mar. 2015. 264

789. S. Head and J. Nelson. Data rights valuation in software acquisitions. Technical Report DRM-2012-001825-Final, CNA Analysis & Solutions, Sept. 2012. 66

790. A. Heathcote, S. Brown, and D. J. K. Mewhort. The power law repealed: The case for an exponential law of practice. *Psychonomic Bulletin & Review*, 7(2):185–207, Apr. 2000. 33

791. R. Heeks. The uneven profile of Indian software exports. Working Paper No. 3, University of Manchester, Oct. 1998. 58

792. J. Heer and M. Bostock. Crowdsourcing graphical perception: Using Mechanical Turk to assess visualization design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI 2010, pages 203–212, Apr. 2010. 221

793. K. Heinze, N. Claussen, and V. LaBolle. Management of computer programming for command and control systems A survey. Technical Memorandum TM-903/000/02, System Development Corporation, Santa Monica, May 1963. 55

794. D. H. Helmer, S. Mackay, K. Selvey-Clinton, R. Yoon, and H. Furukawa. Worldwide capital and fixed assets guide 2016. Technical Report EYG no. DL1528, EYGM Limited, 2016. 80

795. D. R. Helsel. *Statistics for Censored Environmental Data using Minitab and R*. John Wiley & Sons, second edition, 2012. 332

796. A. Hemel and R. Koschke. Reverse engineering variability in source code using clone detection-A case study for Linux variants of consumer electronic devices. In *19th Working Conference on Reverse Engineering*, WCRE'12, pages 357–366, Oct. 2012. 93

797. M. Hendrickson. 2010 state of the computer book market, Feb. 2011. http://radar.oreilly.com/2011/02/2010-book-market-1.html. 110

798. A. Henik and J. Tzelgov. Is three greater than five: The relation between physical and semantic size in comparison tasks. *Memory & Cognition*, 10(4):389–395, 1982. 48

799. J. Henrich. How adaptive cultural processes can produce maladaptive losses–The Tasmanian case. *American Antiquity*, 69(2):197–214, Apr. 2004. 73

800. J. Henrich. The evolution of costly displays, cooperation and religion: credibility enhancing displays and their implications for cultural evolution. *Evolution and Human Behavior*, 30(4):244–260, July 2009. 70

801. J. Henrich, M. Chudek, and R. Boyd. The big man mechanism: how prestige fosters cooperation and creates prosocial leaders. *Philosophical Transactions of The Royal Society B*, 370(1683), Dec. 2015. 115

802. J. Henrich and F. J. Gil-White. The evolution of prestige Freely conferred deference as a mechanism for enhancing the benefits of cultural transmission. *Evolution and Human Behavior*, 22(3):165–196, May 2001. 71

803. J. Henrich, S. J. Heine, and A. Norenzayan. The weirdest people in the world? Working Paper No. 139, German Data Forum (RatSWD), Apr. 2010. 19

804. J. Henrich and R. McElreath. Are peasants risk-averse decision makers? *Current Anthropology*, 43(1):172–181, Feb. 2002. 50

805. I. Heras-Saizarbitoria and O. Boiral. Symbolic adoption of ISO 9000 in small and medium-sized enterprises: The role of internal contingencies. *International Small Business Journal*, 33(3):299–320, May 2015. 145

806. S. Herculano-Houzel. The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost. *PNAS*, 109(1):10661–10668, June 2012. 17

807. F. Hermans and E. Murphy-Hill. Enron's spreadsheets and related emails: A dataset and analysis. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, ICSE'15, pages 7–16, May 2015. 175

808. T. Herr, B. Schneier, and C. Morris. Taking stock: Estimating vulnerability rediscovery. Paper, Belfer Center for Science and International Affairs, Harvard Kennedy School, Oct. 2017. 155

809. E. Herrmann, J. Call, M. V. Hernández-Lloreda, B. Hare, and M. Tomasello. Humans have evolved specialized skills of social cognition: The cultural intelligence hypothesis. *Science*, 317(5843):1360–1366, Sept. 2007. 50, 71

810. R. Hersh. *18 Unconventional Essays on the Nature of Mathematics*. Springer, 2006. 145

811. K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE'13, pages 392–401, May 2013. 12, 145, 148, 375

812. K. Herzig and A. Zeller. Untangling changes. Submitted to MSR 2013, 2013. 375

813. T. Hesterberg. What teachers should know about the bootstrap: Resampling in the undergraduate statistics curriculum. In *eprint arXiv:stat.OT/1411.5279*, Nov. 2014. 264

814. R. J. Heuer, Jr. *Psychology of Intelligence Analysis*. Central Intelligence Agency, 1999. 209

815. M. Hicks, C. O'Malley, S. Nichols, and B. Anderson. Comparison of 2D and 3D representations for visualising telecommunication usage. *Behaviour & Information Technology*, 22(3):185–201, May 2003. 221

816. E. T. Higgins. Value from regulatory fit. *Current Directions in Psychological Science*, 14(4):209–213, 2005. 22

817. N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 1996. 143

818. M. Hilbert and P. López. Supporting online material for: The world's technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, Apr. 2011. 90

819. B. M. Hill and A. Monroy-Hernández. A longitudinal dataset of five years of public activity in the Scratch online community. *Scientific Data*, 4(170002), Jan. 2017. 175

820. T. P. Hill. An evolutionary theory for the variability hypothesis. In *eprint arXiv:q-bio.PE/1703.04184*, Aug. 2018. 19

821. T. T. Hills, P. M. Todd, and M. N. Jones. Foraging in semantic fields: How we search through memory. *Topics in Cognitive Science*, 7(3):513–534, July 2015. 31

822. D. J. Hilton. The social context of reasoning: Conversational inference and rational judgment. *Psychological Bulletin*, 118(2):248–271, 1995. 42

823. A. Hindle, M. W. Godfrey, and R. C. Holt. Reading beside the lines: Indentation as a proxy for complexity metrics. In *The 16th IEEE International Conference on Program Comprehension*, ICPC 2008, pages 133–142, June 2008. 325

824. T. Hirao, A. Ihara, Y. Ueda, P. Phannachitta, and K. ichi Matsumoto. The impact of a low level of agreement among reviewers in a code review process. In *IFIP International Conference on Open Source Systems*, OSS 2016, pages 97–110, May-June 2016. 165

825. S. C. Hirtle and J. Jonides. Evidence for hierarchies in cognitive maps. *Memory & Cognition*, 13(3):208–217, 1985. 48

826. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, Oct. 1969. 144

827. M. Hocko and T. Kalibera. Reducing performance non-determinism via cache-aware page allocation strategies. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, WOSP/SIPEW'10, pages 223–234, Jan. 2010. 368

828. A. Höfer. Exploratory comparison of expert and novice pair programmers. *Computing and Informatics*, 29(1):73–91, 2010. 300

829. E. Hoffer. *The True Believer: Thoughts on the Nature of Mass Movements*. HarperPerennial, 1951. 95

830. D. D. Hoffman. *Visual Intelligence: How We Create What We See*. W. W. Norton, 2000. 17, 24

831. R. Hofman. *Behavioral Products Quality Assessment Model on the Software Market*. PhD thesis, Poznan University of Economics, Oct. 2011. 135

832. J. Hofmeister, J. Siegmund, and D. V. Holt. Shorter identifier names take longer to comprehend. In *24th International Conference on Software Analysis, Evolution and Reengineering*, SANER 2017, pages 217–227, Feb. 2017. 191

833. G. Hofstede. *Culture's Consequences: International Differences in Work-Related Values*. Sage Publications, abridged edition, 1984. 69

834. R. M. Hogarth and H. J. Einhorn. Order effects in belief updating: The belief-adjustment model. *Cognitive Psychology*, 24(1):1–55, Jan. 1992. 36, 37

835. R. M. Hogarth, C. R. M. McKenzie, B. J. Gibbs, and M. A. Marquis. Learning from feedback: Exactness and incentives. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 17(4):734–752, 1991. 37

836. B. D. Holbrook and W. S. Brown. A history of computing research at Bell Laboratories (1937-1975). Computing Science Technical Report No. 99, AT&T Bell Laboratories, 1982. 88

837. M. Holdway. An alternative methodology: Valuing quality change for microprocessors in the PPI. In *Issues in Measuring Price Change and Consumption*. Bureau of Labor Statistics, June 2000. 5

838. W. B. Holland. Soviet cybernetics technology: viii. Report on the algorithmic language ALGEC (final version). Research Memorandum RM-5136-PR, The RAND Corporation, Dec. 1966. 103

839. J. K. Hollmann. Estimate accuracy: Dealing with reality. *Cost Engineering Journal*, 54(6):17–27, Nov.-Dec. 2012. 121

840. A. A. Hook, B. Brykczynski, C. W. McDonald, S. H. Nash, and C. Youngblut. A survey of computer programming languages currently used in the Department of Defense. IDA Paper P-3054, Institute for Defense Analyses, Jan. 1995. 110

841. R. Hoosain. Correlation between pronunciation speed and digit span size. *Perception and Motor Skills*, 55:1128–1128, 1982. 358

842. R. Hoosain and F. Salili. Language differences, working memory, and mathematical ability. In M. M. Grunberg, P. E. Morris, and R. N. Sykes, editors, *Practical aspects of memory: Current research and issues*, volume 2, pages 512–517. John Wiley & Sons, Inc, 1988. 29

843. M. Hoppe and S. Hanenberg. Do developers benefit from generic types? An empirical comparison of generic and raw types in Java. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA'13, pages 457–474, Oct. 2013. 201

844. W. Hoppitt and K. N. Laland. *Social Learning: An Introduction to Mechanisms, Methods, and Models*. Princeton University Press, July 2013. 71

845. W. Hordijk, M. L. Ponisio, and R. Wieringa. Harmfulness of code duplication a structured review of the evidence. In 13^{th} *International Conference on Evaluation and Assessment in Software Engineering*, EASE'09, pages 88–97, Apr. 2009. 77

846. V. Horký. *Performance Awareness in Agile Software Development*. PhD thesis, Charles University, Faculty of Mathematics and Physics, Mar. 2018. 134

847. Z. Horne, M. Muradoglu, and A. Cimpian. Explanation as a cognitive process. *Trends in Cognitive Sciences*, 23(3):187–199, Mar. 2019. 181

848. M. R. Horton. *Portable C Software*. Prentice-Hall, Inc, Upper Saddle River, NJ 07458, USA, 1990. 179

849. S. Hossenfelder. *Lost in Math: How Beauty Leads Physics Astray*. Basic Books, June 2018. 10

850. D. A. Hounshell. *From the American System to Mass Production 1800-1932: The Development of Manufacturing Technology in the United States*. The Johns Hopkins University Press, 1984. 97

851. A. D. Householder and J. M. Foote. Probability-based parameter selection for black-box fuzz testing. Technical Note CMU/SEI-2012-TN-019, Software Engineering Institute, Carnegie Mellon University, Aug. 2012. 169

852. M. W. Howard and M. J. Kahana. Context variability and serial position effects in free recall. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 25(4):923–941, 1999. 32

853. J. Howison and J. B. Herbsleb. Incentives and integration in scientific software production. In *Proceedings of the 2013 conference on Computer supported cooperative work*, CSCW'13, pages 459–470, Mar. 2013. 70

854. L. Hribar, S. Bogovac, and Z. Marinčić. Implementation of fault slip through in design phase of the project. In *miproBIS 2008: International Conference on Business Intelligence Systems*, May 2008. 164

855. H. Hsu. *The Appsmiths: Community, Identity, Affect and Ideology Among Cocoa Developers From NeXT to Iphone*. PhD thesis, Graduate School of Cornell University, May 2015. 70

856. http archive. `https:httparchive.org`, July 2018. 94

857. X. Huang, J. Xie, N. O. Otecko, and M. Peng. Accessibility and update status of published software: Benefits and missed opportunities. *Frontiers in Research Metrics and Analytics*, 2(doi.org/10.3389/frma.2017.00001), Feb. 2017. 140

858. B. A. Huberman. The dynamics of organizational learning. *Computational & Mathematical Organization Theory*, 7(2):145–153, Aug. 2001. 72

859. H. Huijgens and R. van Solingen. Measuring best-in-class software releases. In *Joint Conference of the 23rd International Workshop on Software Measurement and the 8th International Conference on Software Process and Product Measurement*, pages 137–146, Oct. 2013. 125

860. H. Huijgens and F. Vogelezang. Do estimators learn? On the effect of a positively skewed distribution of effort data on software portfolio productivity. Technical Report TUD-SERG-2016-004, Delft University of Technology, 2016. 73

861. J. C. Hull. *Options, Futures, and other Derivatives*. Pearson, seventh edition, Oct. 2010. 61

862. C. Hulme, S. Maughan, and G. D. A. Brown. Memory for familiar and unfamiliar words: Evidence for a long-term memory contribution to short-term memory span. *Journal of Memory and Language*, 30(6):685–701, 1991. 30

863. C. R. Hulten. Decoding Microsoft: Intangible capital as a source of company growth. Working Paper 15799, National Bureau of Economic Research, USA, Mar. 2010. 78

864. R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. MAO-an extensible micro-architectural optimizer. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'11, pages 1–10, Apr. 2011. 365

865. S. Hunold and A. Carpen-Amarie. MPI benchmarking revisited: Experimental design and reproducibility. In *eprint arXiv:cs.DC/1505.07734v3*, Sept. 2015. 364

866. S. Hunold, A. Carpen-Amarie, and J. L. Träff. Reproducible MPI micro-benchmarking isn't as easy as you think. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA'14, pages 69–76, Sept. 2014. 241

867. E. Hunt. The Whorfian hypothesis: A cognitive psychology perspective. *Psychological Review*, 98(3):377–389, July 1991. 196

868. J. E. Hunter, F. L. Schmidt, and M. K. Judiesch. Individual differences in output variability as a function of job complexity. *Journal of Applied Psychology*, 75(1):28–42, Feb. 1990. 55

869. M. J. Hurlstone, G. J. Hitch, and A. D. Baddeley. Memory for serial order across domains: An overview of the literature and directions for future research. *Psychonomic Bulletin & Review*, 140(2):229–373, Mar. 2014. 32

870. A. A. Hwang, I. A. Stefanovici, and B. Schroeder. Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 111–122, Mar. 2012. 162

871. S. Ibba, F. E. Pani, J. G. Stockton, G. Barabino, M. Marchesi, and D. Tigano. Incidence of predatory journals in computer science literature. *Library Review*, 66(6-7):505–522, Sept. 2017. 9

872. IBM. Specifications for the IBM mathematical FORmula TRANSlating system, FORTRAN. Programming Research Group, Applied Science Division, International Business Machines Corporation, Nov. 1954. 109

873. R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of Lua. In *Proceedings of the third ACM SIGPLAN conference on History of Programming Languages*, HOPL III, pages 1–26, June 2007. 89

874. J. Iivonen. Identifying and characterizing highly performing testers-A case study in three software product companies. Thesis (m.s.), Helsinki University of Technology, Department of Computer Science and Engineering, Oct. 2009. 55, 56

875. S. Ikeda, A. Ihara, R. G. Kula, and K. Matsumoto. An empirical study of README contents for JavaScript packages. *IEICE Transactions on Information & Systems*, E102-D(2):280–288, Feb. 2019. 174

876. Y. Ikutani, T. Kubo, S. Nishida, H. Hata, K. Matsumoto, K. Ikeda, and S. Nishimoto. Expert programmers have fine-tuned cortical representations of source code. In *bioRxiv doi: 10.1101/2020.01.28.923953*, Jan. 2020. 173

877. I. Imbo and J.-A. LeFevre. Cultural differences in complex addition: Efficient Chinese versus adaptive Belgians and Canadians. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 35(6):1465–1476, Nov. 2009. 46

878. I. Imbo, A. Vandierendonck, and E. Vergauwe. The role of working memory in carrying and borrowing. *Psychological Research*, 71(4):467–483, July 2007. 46

879. L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 435–445, June 2014. 170

880. Intel. 6th generation Intel processor family. Specification update 332689-010EN, Intel Corporation, Apr. 2017. 157

881. International telecommunication union. website, July 2012. http://www.itu.int. 158

882. J. P. A. Ioannidis. Contradicted and initially stronger effects in highly cited clinical research. *JAMA*, 294(2):218–228, July 2005. 10

883. J. P. A. Ioannidis. Why most published research findings are false. *PLoS Medicine*, 2(8):e124, Aug. 2005. 264

884. A. Iosup, M. Jan, O. Sonmez, and D. H. J. Epema. On the dynamic resource availability in grids. Rapport de recherche no 6172, Institut National de Recherche en Informatique et en Automatique, Apr. 2007. 163

885. G. Irlam. Unix file size survey-1993. http://www.base.com/gordoni/ufs93.html, Sept. 1993. 242, 243

886. F. Irving. Github users since service started. https://classic.scraperwiki.com/scrapers/github_users_each_year, Mar. 2016. 83

887. ISO. *ISO/IEC Guide 25:1990 General requirements for the competence of calibration and testing laboratories*. International Organization for Standardization, 1990. 168

888. ISO. *ISO/IEC 9945:2008 Information technology – Portable Operating System Interface (POSIX®)*. International Organization for Standardization, 2008. 111, 158

889. ISO SC22. *ISO/IEC 18009:1999 Information technology – Programming languages – Ada: Conformity assessment of a language processor*. International Organization for Standardization, 1990. Last reviewed and confirmed in 2015. 168

890. ISO SC22. *ISO/IEC 13210:1999 Information technology – Requirements and guidelines for test methods specifications and test method implementation for measuring conformance to POSIX standards*. International Organization for Standardization, 1999. 168

891. A. Israeli and D. G. Feitelson. The Linux kernel as a case study in software evolution. *Journal of Systems and Software*, 83(3):485–501, Mar. 2010. 194, 286, 287, 292, 313, 314

892. R. K. Iyer, S. E. Butner, and E. J. McCluskey. An exponential failure/load relationship: Results of a multi-computer statistical study. Technical Report #CRC-81-6, Computer Systems Laboratory, Stanford University, Aug. 1981. 147

893. J. L. C. Izquierdo and J. Cabot. A survey of software foundations in Open Source. In *eprint arXiv:cs.SE/2005.10063.pdf*, May 2020. 89

894. M. Y. Jaber. Learning and forgetting models and their applications. In A. B. Badiru, editor, *Handbook of Industrial and Systems Engineering*, chapter 30. CRC Press-Taylor & Francis Group, Dec. 2005. 72

895. A. N. Jackson. Formats over time: Exploring UK web history. In *eprint arXiv:cs.DL/1210.1714v1*, Oct. 2012. 107

896. P. Jackson. Opus development postmortem: Joe comes, riley paint, inc., skeffingon's formal wear, inc., patricia anne larsen vs. microsoft corporation. Plaintiff's Exhibit 8875, IOWA District Court for Polk County, Dec. 1989. 137, 138

897. J. Jacobs and B. Rudis. *Data-Driven Security: Analysis, Visualization and Dashboards*. John Wiley & Sons, Inc, 2014. 209, 381

898. R. Jaeschke. *Portability and the C Language*. Hayden Books, 4300 West 62nd Street, Indianapolis, IN 46268, USA, 1989. 179

899. L. R. Jager and J. T. Leek. Empirical estimates suggest most published research is true. *Biostatistics*, 15(1):1–12, 2014. 264

900. B. Jamtveit, E. Jettestuen, and J. Mathiesen. Scaling properties of European research units. *PNAS*, 106(32):13160–13163, Aug. 2009. 104

901. A. R. Jansen. *Encoding and Parsing of Algebraic Expressions by Experienced Users of Mathematics*. PhD thesis, School of Computer Science and Software Engineering, Monash University, Jan. 2002. 27

902. A. R. Jansen, K. Marriott, and G. W. Yelland. Parsing of algebraic expressions by experienced users of mathematics. *European Journal of Cognitive Psychology*, 19(2):286–320, 2007. 27

903. C. J. M. Jansen and M. M. W. Pollmann. On round numbers: Pragmatic aspects of numerical expressions. *Journal of Quantitative Linguistics*, 8(3):187–201, 2001. 47

904. Y. Jansen and K. Hornbæk. A psychophysical investigation of size as a physical variable. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):479–488, Jan. 2016. 221

905. J. J. Jenkins. Remember that old theory of memory? Well, forget it! *American Psychologist*, 29(11):785–795, 1974. 183

906. J. Jiang, D. Lo, J. He, X. Xia, P. S. Kochhar, and L. Zhang. Why and how developers fork what from whom in GitHub. *Empirical Software Engineering*, 22(1):547–578, Feb. 2017. 92

907. Y. Jiang, B. Adams, and D. M. German. Will my patch make it? And how fast? Case study on the Linux kernel. In *10th Working Conference on Mining Software Repositories*, MSR'13, pages 101–110, May 2013. 141

908. D. D. P. Johnson, N. B. Weidmann, and L.-E. Cederman. Fortune favours the bold: An agent-based model reveals adaptive advantages of overconfidence in war. *PLoS ONE*, 6(6):e20851, Apr. 2011. 53

909. J. A. Johnson. Measuring thirty facets of the Five Factor model with a 120-item public domain inventory: Development of the IPIP-NEO-120. *Journal of Research in Personality*, 51:78–89, Aug. 2014. 50

910. L. Johnson. Applied data research inc. (ADR). Technical report, Computer History Museum, Feb. 2010. 103

911. P. M. Johnson and A. M. Disney. A critical analysis of PSP data quality: Results from a case study. *Empirical Software Engineering*, 4(4):317–349, Dec. 1999. 374

912. W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors*. Morgan Kaufmann Publishers, Inc, 1986. 195

913. C. I. Jones. The facts of economic growth. In J. B. Taylor and H. Uhlig, editors, *Handbook of Macroeconomics, Volume 2A*, chapter 1, pages 3–69. Elsevier B. V., Nov. 2016. 5

914. D. Jones. Why userspace sucks–or 101 really dumb things your app shouldn't do. In *Proceedings of the Linux Symposium*, Volume One, pages 441–450, July 2006. 367

915. D. M. Jones. Who guards the guardians? www.knosof.co.uk/whoguard.html, 1992. 158

916. D. M. Jones. *The Open Systems Portability Checker Reference Manual*. Knowledge Software Ltd, ??? edition, May 1999. 158

917. D. M. Jones. The 7±2 urban legend. MISRA C 2002 conference http://www.knosof.co.uk/cbook/misart.pdf, Oct. 2002. 28, 358

918. D. M. Jones. Memory for a short sequence of assignment statements (part 2 of 2). *C Vu*, 17(1):34–37, Feb. 2005. 177

919. D. M. Jones. The new C Standard: An economic and cultural commentary. Knowledge Software, Ltd, 2005. 8, 90, 111, 132, 158, 170, 189, 190, 195, 196, 199, 202, 203, 206, 211, 222, 246, 298, 358, 381

920. D. M. Jones. Developer beliefs about binary operator precedence. *C Vu*, 18(4):14–21, Aug. 2006. 35, 36, 50, 56, 202, 350, 351, 358, 371, 372

921. D. M. Jones. Operand names influence operator precedence decisions. *C Vu*, 20(1):5–11, Feb. 2008. 50, 189, 371, 372

922. D. M. Jones. Deciding between `if` and `switch` when writing code. *C Vu*, 21(5):14–20, Nov. 2009. 199

923. D. M. Jones. Developer characterization of data structure fields decisions. *C Vu*, 20(6):14–18, Jan. 2009. 41, 56, 246, 247, 349, 350, 371, 372

924. D. M. Jones. Birth month of compiler writers. blog: The Shape of Code, Feb. 2012. http://shape-of-code.coding-guidelines.com. 342

925. D. M. Jones. Effects of risk attitude on recall of assignment statements. *C Vu*, 23(6):19–22, Jan. 2012. 50

926. D. M. Jones. Tag data extracted from stack overflow website. url-https://stackoverflow.com, July 2019. 110

927. D. M. Jones. Code & data used in: Evidence-based software engineering: based on the publicly available data. http://www.github.com/Derek-Jones/ESEUR, 2020. 2

928. D. M. Jones and R. Borgatti. The Renzo Pomodoro dataset: a conversation. http://www.github.com/Derek-Jones/renzo-pomodoro, Dec. 2019. 139

929. D. M. Jones and S. Cullum. A conversation around the analysis of the SiP effort estimation dataset. In *eprint arXiv:cs.SE/1901.01621*, Jan. 2019. 122, 123, 133, 134, 137

930. M. N. Jones and D. J. K. Mewhort. Case-sensitive letter and bigram frequency counts from large-scale English corpora. *Behavior Research Methods, Instruments, & Computers*, 36(3):388–396, 2004. 195

931. R. Jongeling, P. Sarkar, S. Datta, and A. Serebrenik. On negative results when using sentiment analysis tools for software engineering research. *Empirical Software Engineering*, 22(5):2543–2584, Oct. 2017. 346

932. M. R. Jongerden. *Model-based energy analysis of battery powered systems*. PhD thesis, Centre for Telematics and Information Technology, University of Twente, Dec. 2010. 364

933. M. Jørgensen. An empirical study of software maintenance tasks. *Software Maintenance: Research and Practice*, 7(1):27–48, Jan. 1995. 300

934. M. Jørgensen. Regression models of software development effort estimation accuracy and bias. *Empirical Software Engineering*, 9(4):297–394, Dec. 2004. 122

935. M. Jørgensen. Better selection of software providers through trialsourcing. *IEEE Software*, 33(5):48–53, Sept.-Oct. 2016. 122, 126

936. M. Jørgensen. A survey on the characteristics of projects with success in delivering client benefits. *Information and Software Technology*, 78:83–94, Oct. 2016. 131

937. M. Jørgensen. Unit effects in software project effort estimation: Work-hours gives lower effort estimates than workdays. *Journal of Systems and Software*, 117:274–281, July 2016. 48

938. M. Jørgensen and G. J. Carelius. An empirical study of software project bidding. *IEEE Transactions on Software Engineering*, 30(12):953–969, Dec. 2004. 119, 271

939. M. Jørgensen, T. Dybå, K. Liestøl, and D. I. K. Sjøberg. Incorrect results in software engineering experiments: How to improve research practices. *Journal of Systems and Software*, 116:133–145, June 2016. 264

940. M. Jørgensen and S. Grimstad. Software development estimation biases: The role of interdependence. *IEEE Transactions on Software Engineering*, 38(3):677–693, May 2012. 19, 23

941. M. Jørgensen, U. Indahl, and D. I. K. Sjøberg. Software effort estimation by analogy and "regression toward the mean". *Journal of Systems and Software*, 68(3):253–262, Dec. 2003. 285

942. M. Jørgensen and K. Moløkken. Eliminating over-confidence in software development effort estimates. In F. Bomarius and H. Iida, editors, *Product Focused Software Process Improvement*, volume 3009 of *Lecture Notes in Computer Science*, pages 174–184. Springer Berlin Heidelberg, Apr. 2004. 272

943. M. Jørgensen and K. Moløkken. Reasons for software effort estimation error: Impact of respondent role, information collection approach, and data analysis method. *IEEE Transactions on Software Engineering*, 30(12):993–1007, Dec. 2004. 21, 125

944. M. Jørgensen and K. Moløkken. How large are software cost overruns? A review of the 1994 CHAOS report. *Journal Information and Software Technology*, 48(4):297–301, Apr. 2006. 118

945. M. Jørgensen and D. I. K. Sjøberg. The impact of customer expectation on software development effort estimates. *International Journal of Project Management*, 22(4):317–325, May 2004. 22, 123

946. M. Jørgensen and D. I. K. Sjøberg. Learning from experience in a software maintenance environment. *Journal of Computer Science*, 1(4):538–542, Apr. 2005. 293

947. D. Joseph, W. F. Boh, S. Ang, and S. A. Slaughter. The career paths less (or more) travelled: A sequence analysis of IT career histories, mobility patterns, and career success. *MIS Quarterly*, 36(2):427–452, June 2012. 105

948. J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim. Fuzzification: Anti-fuzzing techniques. In *28th USENIX Security Symposium*, SEC'19, pages 1913–1930, Aug. 2019. 169

949. S. Kahrs. Mistakes and ambiguities in the definition of standard ML. LFCS report ECS-LFCS-93-257, University of Edinburgh, Scotland, Apr. 1993. 161

950. J. W. Kalat. *Biological Psychology*. Wadsworth, seventh edition, 2001. 18

951. T. Kalibera, L. Bulej, and P. Tůma. Benchmark precision and random initial state. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, SPECTS 2005, pages 853–862. Society for Modeling and Simulation (SCS), July 2005. 368

952. A. Kaltenbrunner, V. Gómez, A. Moghnieh, R. Meza, J. Blat, and V. López. Homogeneous temporal activity patterns in a large online communication space. In *eprint arXiv:cs.NI/0708.1579*, Aug. 2007. 242

953. C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel. Distance-based sampling of software configuration spaces. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE'19, pages 1084–1094, May 2019. 168

954. D. Kaminsky, M. Eddington, and A. Cecchetti. Showing how security has (and hasn't) improved, after ten years of trying. CanSecWest Applied Security Conference, Dec. 2011. 155, 156, 157

955. T. Kamiya. How code skips over revisions. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC 2011, pages 69–70, May 2011. 196

956. V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg. A systematic review of effect size in software engineering experiments. *Information and Software Technology*, 49(11-12):1073–1086, Apr. 2007. 6

957. P. Kampstra and C. Verhoef. Benchmarking the expected loss of a federal IT portfolio. ???, July 2009. 281

958. P. Kampstra and C. Verhoef. Reliability of function point counts. http://www.cs.vu.nl/~x/rofpc/rofpc.pdf, 2009. 125

959. T. Kanda, T. Ishio, and K. Inoue. Approximating the evolution history of software from source code. *IEICE Transactions on Information & Systems*, E98-D(6):1185–1193, June 2015. 348

960. C. Kaner. Liability for defective documentation. In *Proceedings of the 21st annual international conference on Documentation*, SIGDOC'03, pages 192–197, Oct. 2003. 135, 161

961. C. Kaner and D. Pels. *Bad Software: What To Do When Software Fails*. John Wiley & Sons, Inc, 1998. 150

962. Y. Kang, B. Ray, and S. Jana. APEx: Automated inference of error specifications for C APIs. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 472–482, Sept. 2016. 170

963. S. J. Karau and K. D. Williams. Social loafing: A meta-analytic review and theoretical integration. *Journal of Personality and Social Psychologs*, 65(4):681–706, Oct. 1993. 68, 76

964. D. Karlis and E. Xekalaki. Mixed poisson distributions. *International Statistical Review*, 73(1):35–58, Apr. 2005. 235

965. J. Karlsson and K. Ryan. A cost-value approach for prioritizing requirements. *IEEE Software*, 14(5):67–74, Sept. 1997. 132

966. D. S. Katz, K. McHenry, C. Reinking, and R. Haines. Research software development & management in universities: Case studies from Manchester's RSDS group, Illinois' NCSA, and Notre Dame's CRC. In *eprint arXiv:cs.SE/1903.00732*, Mar. 2019. 105

967. G. Kawasaki. *Selling the Dream: How to Promote Your Product, Company, or Ideas–and Make a Difference–Using Everyday Evangelism*. HarperBusiness, Jan. 1991. 70, 96

968. D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE'11, pages 351–360, May 2011. 135

969. M. Kazandjieva, B. Heller, O. Gnawali, P. Levis, and C. Kozyrakis. Green enterprise computing data: Assumptions and realities. In *Proceedings of the 2012 International Green Computing Conference*, IGCC'12, pages 1–10, June 2012. 91

970. F. C. Keil. Explanation and understanding. *Annual Review of Psychology*, 57:227–254, Jan. 2006. 182

971. M. Keil and D. Robey. Blowing the whistle on troubled software projects. *Communications of the ACM*, 44(4):87–93, Apr. 2001. 130

972. P. Keil, J. M. Bennett, B. Bourgeois, G. E. Garcá-Peña, A. A. M. MacDonald, C. Meyer, K. S. Ramirez, and B. Yguel. From computer operating systems to biodiversity: co-emergence of ecological and evolutionary patterns. *PNAS*, 4:e2367, Aug. 2016. 93

973. C. F. Kemerer. An empirical validation of software cost estimation models. *Communications of the ACM*, 30(5):416–429, May 1987. 124

974. Z. Kenessey. The primary, secondary, tertiary and quaternary sectors of the economy. *The Review of Income and Wealth*, 33(4):359–385, Dec. 1987. 57

975. D. O. Kennedy and A. B. Scholey. Glucose administration, heart rate and cognitive performance: effects of increasing mental effort. *Psychopharmacology*, 149(1):63–71, May 2000. 55

976. E. Keogh and A. Mueen. Time series data mining using the matrix profile: A unifying view of motif discovery, anomaly detection, segmentation, classification, clustering and similarity joins. Tutorial at KDD 2017, Aug. 2017. 331

977. B. W. Kernighan and R. Pike. *The Practice of Programming*. Addison–Wesley, 1999. 179

978. N. L. Kerr. HARKing: Hypothesizing After the Results are Known. *Personality and Social Psychology Review*, 2(3):196–217, Aug. 1998. 11

979. E. Keuleers, P. Lacey, K. Rastle, and M. Brysbaert. The British lexicon project: Lexical decision data for 28,730 monosyllabic and disyllabic English words. *Behavior and Research Methods*, 44(1):287–304, Mar. 2012. 33

980. H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan. Prioritizing the devices to test your app on: A case study of Android game apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 610–620, Nov. 2014. 80

981. L. M. Khan. Amazon's antitrust paradox. *The Yale Law Journal*, 126(3):710–805, Jan. 2017. 98

982. M. W. Khaw, L. Stevens, and M. Woodford. Discrete adjustment to a changing environment: Experimental evidence. *Journal of Monetary Economics*, 91(C):88–103, 2017. 49

983. P.-V. Khuong and P. Morin. Array layouts for comparison-based searching. In *eprint arXiv:cs.DS/1509.05053*, Mar. 2017. 367

984. P. D. Killworth and H. R. Bernard. Informant accuracy in social network data. *Human Organization*, 35(3):269–286, Sept.-Nov. 1976. 354

985. D. Kim, E. Murphy-Hill, C. Parnin, C. Bird, and R. Garciad. The reaction of open-source projects to new language features: An empirical study of C# generics. *The Journal of Object Technology*, 12(4):1–26, Nov. 2013. 181

986. H. Kim. *Informed Storage Management for Mobile Platforms*. PhD thesis, College of Computing, Georgia Institute of Technology, Dec. 2012. 366

987. J. D. Kim. Startup acquisitions as a hiring strategy: Worker choice and turnover. SSRN Working Paper 3252784, Wharton School, University of Pennsylvania, Mar. 2020. 105

988. J. Y. Kim, S. Shepherd, T. H. Campbell, and A. C. Kay. Understanding contemporary forms of exploitation: Attributions of passion serve to legitimize the poor treatment of workers. *Journal of Personality and Social Psychology: Interpersonal Relations and Group Processes*, 118(1):121–148, Jan. 2020. 67

989. S. Kim. The classification of information and communication technology investment in financial accounting. Thesis (m.s.), School of Information Technologies, University of Sydney, 2013. 80

990. K. Kina, M. Tsunoda, H. Hata, H. Tamada, and H. Igaki. Analyzing the decision criteria of software developers based on prospect theory. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, SANER'16, pages 644–648, Mar. 2016. 53

991. D. King and C. Janiszewski. The sources and consequences of the fluent processing of numbers. *Journal of Marketing Research*, XLVIII(2):327–341, 2011. 47

992. J. King and M. A. Just. Individual differences in syntactic processing: The role of working memory. *Journal of Memory and Language*, 30:580–602, 1991. 30

993. W. Kintsch. *Comprehension: A paradigm for cognition*. Cambridge University Press, 1998. 185

994. W. Kintsch and J. Keenan. Reading rate and retention as a function of the number of propositions in the base structure of sentences. *Cognitive Psychology*, 5(3):257–274, Nov. 1973. 183

995. W. Kintsch, E. Kozminsky, W. J. Streby, G. McKoon, and J. M. Keenan. Comprehension and recall of text as a function of content variables. *Journal of Verbal Learning and Verbal Behavior*, 14(2):196–214, Apr. 1975. 183

996. W. Kintsch, T. S. Mandel, and E. Kozminsky. Summarizing scrambled stories. *Memory & Cognition*, 5(5):547–552, 1977. 186

997. K. N. Kirby and R. J. Herrnstein. Preference reversals due to myopic discounting of delayed reward. *Psychological Science*, 6(2):83–89, Mar. 1995. 54

998. D. Kirsh and P. Maglio. On distinguishing epistemic from pragmatic action. *Cognitive Science*, 18(4):513–549, Oct. 1994. 20

999. L. B. Kish. Moore's law and the energy requirement of computing versus performance. *IEE Proceedings-Circuits, Devices and Systems*, 151(2):190–194, Apr. 2004. 161

1000. J. V. Kistowski, H. Block, J. Beckett, K.-D. Lange, J. A. Arnold, and S. Kounev. Analysis of the influences on server power consumption and energy efficiency for CPU-intensive workloads. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE'15, pages 223–234, Jan. 2015. 251

1001. S. Kitayama and M. Karasawa. Implicit self-esteem in Japan: Nameletters and birthday numbers. *Personality & Social Psychology Bulletin*, 23(7):736–742, 1997. 190

1002. B. Kitchenham, S. L. Pfleeger, B. McColl, and S. Eagan. An empirical study of maintenance and development estimation accuracy. *The Journal of Systems and Software*, 64(1):57–77, Oct. 2002. 122

1003. B. A. Kitchenham and N. R. Taylor. Software project development cost estimation. *The Journal of Systems and Software*, 5(4):267–278, Nov. 1985. 129

1004. A. Kivi, T. Smura, and J. Töyli. Technology product evolution and the diffusion of new product features. *Technological Forecasting & Social Change*, 79(1):107–126, Jan. 2012. 82

1005. D. Klahr, W. G. Chase, and E. A. Lovelace. Structure and process in alphabetic retrieval. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 9(3):462–477, 1983. 32

1006. K. C. Klauer, J. Musch, and B. Naumer. On belief bias in syllogistic reasoning. *Psychological Review*, 107(4):852–884, Oct. 2000. 43

1007. J. Klayman and Y.-W. Ha. Confirmation, disconfirmation, and information in hypothesis testing. *Psychological Review*, 94(2):211–228, Apr. 1987. 23

1008. B. Klein. The decision making problem in development. In Universities-National Bureau Committee for Economic Research, Committee on Economic Growth of the Social Science Research Council, editor, *The Rate and Direction of Inventive Activity: Economic and Social Factors*, chapter 19, pages 477–508. Princeton University Press, 1962. 126

1009. S. B. Klein, L. Cosmides, J. Tooby, and S. Chance. Decisions and the evolution of memory: Multiple systems, multiple functions. *Psychological Review*, 109(2):306–329, Apr. 2002. 27

1010. J. Kleinberg and M. Raghu. Team performance with test scores. In *eprint arXiv:cs.DS/1506.00147v2*, Mar. 2018. 137

1011. S. Kleinschmager, R. Robbes, A. Stefik, S. Hanenberg, and É. Tanter. Do static type systems improve the maintainability of software systems? An empirical study. In *20th International Conference on Program Comprehension*, ICPC'12, pages 153–162, June 2012. 201

1012. S. Klepper and K. L. Simons. Technological extinctions of industrial firms: An inquiry into their nature and causes. *Industrial and Corporate Change*, 6(2):379–460, Mar. 1997. 103

1013. P. Klint, D. Landman, and J. Vinju. Exploring the limits of domain model recovery. In *29th IEEE International Conference on Software Maintenance*, ICSM'13, pages 120–129, Sept. 2013. 132

1014. K. E. Knight. Changes in computer performance. *Datamation*, 12(9):40–54, Sept. 1966. 361

1015. K. E. Knight. Evolving computer performance 1963-1967. *Datamation*, 14(1):31–35, Jan. 1968. 6, 361

1016. D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison–Wesley, first edition, 1973. 90

1017. D. E. Knuth. Structure programming with go to statements. *Computing Surveys*, 6(4):261–301, Dec. 1974. 199

1018. D. E. Knuth. The errors of TEX. *Software–Practice and Experience*, 19(7):607–685, 1989. 147

1019. D. Kobak, S. Shpilkin, and M. S. Pshenichnikov. Integer percentages as electoral falsification fingerprints. In *eprint arXiv:stat.AP/1410.6059v4*, June 2016. 382

1020. A. Koenig. *C Traps and Pitfalls*. Addison–Wesley, 1989. 179

1021. P. A. Kolers. Reading A year later. *Journal of Experimental Psychology: Human Learning and Memory*, 2(3):554–565, 1976. 187, 188

1022. P. A. Kolers and D. N. Perkins. Spatial and ordinal components of form perception and literacy. *Cognitive Psychology*, 7(2):228–267, Apr. 1975. 187

1023. J. G. Koomey, S. Berard, M. Sanchez, and H. Wong. Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing*, 33(3):46–54, July-Sept. 2011. 4

1024. A. Koriat. How do we know that we know? The accessibility model of the feeling of knowing. *Psychological Review*, 100(4):609–639, Oct. 1993. 31

1025. A. G. Koru, K. El Emam, D. Zhang, H. Liu, and D. Mathew. Theory of relative defect proneness: Replicated studies on the functional form of the size-defect relationship. *Empirical Software Engineering*, 13(5):473–498, Oct. 2008. 160

1026. J. Kossik. Clark's sector model for US economy 1850-2009. website, 2011. http://www.63alfred.com/whomakesit/clarksmodel.htm. 57

1027. S. M. Kosslyn. *Graph Design for the Eye and Mind*. Oxford University Press, 2006. 224

1028. S. M. Kosslyn and S. P. Shwartz. Empirical constraints on theories of visual imagery. In J. Long and A. D. Baddeley, editors, *Attention and Performance IX*, pages 241–260. Lawrence Erlbaum Associates, 1981. 20

1029. KPMG. Project Tesla due diligence assistance. submitted as evidence in court case, Aug. 2011. 78

1030. P. Kraft. *Programmers and Managers: The Routinization of Computer Programming in the United States*. Springer-Verlag, July 1977. 69, 137

1031. M. Kremer. The O-ring theory of economic development. *The Quarterly Journal of Economics*, 108(3):551–575, Aug. 1993. 137

1032. E. Krevat, J. Tucek, and G. R. Ganger. Disks are like snowflakes: No two are alike. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, May 2013. 366

1033. G. Kroah-Hartman. Linux kernel statistics. website, June 2016. https://www.github.com/gregkh/kernel-history. 283, 309

1034. J. K. Kruschke. Human category learning: Implications for backpropagation models. *Connection Science*, 5(1):3–36, 1993. 34, 35

1035. J. K. Kruschke. Dimensional relevance shifts in category learning. *Connection Science*, 8(2):225–247, June 1996. 34, 35

1036. E. C. Kubie. Recollections of the first software company. *IEEE Annals of the History of Computing*, 16(2):65–71, June 1994. 103

1037. M. Kubovy and M. van den Berg. The whole is equal to the sum of its parts: A probabilistic model of grouping by proximity and similarity in regular patterns. *Psychological Review*, 115(1):131–154, Jan. 2008. 25, 26

1038. T. Kuchta, T. Lutellier, E. Wong, L. Tan, and C. Cadar. On the correctness of electronic documents: studying, finding, and localizing inconsistency bugs in PDF readers and files. *Empirical Software Engineering*, 23(6):3187–3220, Dec. 2018. 107

1039. B. M. Kuhn, Free Software Foundation, Inc., Software Freedom Law Center, A. K. Sebro, Jr., D. Gingerich, and C. Legal. *Copyleft and the GNU General Public License: A Comprehensive Tutorial and Guide*. copyleft.org, 2018. 64

1040. D. R. Kuhn, R. N. Kacker, and Y. Lei. A model for t-way fault profile evolution during testing. In *IEEE International Conference on Software Testing, Verification and Validation Workshops*, ICSTW 2017, pages 162–170, Mar. 2017. 169

1041. M. Kuhrmann, C. Konopka, P. Nellemann, P. Diebold, and J. Münch. Software process improvement: Where is the evidence? In *Proceedings of the 2015 International Conference on Software and System Process*, pages 107–116, Aug. 2015. 7

1042. R. Kumar. The business of scaling. *IEEE Solid-State Circuits Society Newsletter*, 12(1):22–26, 2007. 4

1043. S. Kumar. Enforcing the GNU GPL. *Journal of Law, Technology & Policy*, 2006(1), 2006. 66

1044. G. Kunda. *Engineering Culture: Control and Commitment in a High-Tech Corporation*. Temple University Press, 1992. 103

1045. P. Küngas, S. Vakulenko, M. Dumas, C. Parra, and F. Casati. Reverse-engineering conference rankings: What does it take to make a reputable conference? *Scientometrics*, 96(2):651–665, Aug. 2013. 9

1046. G. Kunst. Language popularity. http://langpop.corger.nl/results, 2013. 288

1047. R. Kurzban, A. Duckworth, J. W. Kable, and J. Myers. An opportunity cost model of subjective effort and task performance. *Behavioral and Brain Sciences*, 36(6):661–679, Dec. 2013. 24

1048. D. S. Kusumo, M. Staples, L. Zhu, and R. Jeffery. Analyzing differences in risk perceptions between developers and acquirers in OTS-based custom software projects using stakeholder analysis. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM'12, pages 69–78, Sept. 2012. 120

1049. A. Kvarven, E. Strømland, and M. Johannesson. Comparing meta-analyses and preregistered multiple-laboratory replication projects. *Nature Human Behaviour*, 4(4):423–434, Apr. 2020. 261

1050. F. Křikava, H. Miller, and J. Vitek. Scala implicits are everywhere: A large-scale study of the use of implicits in the wild. In *eprint arXiv:cs.PL/1908.07883*, Aug. 2019. 193

1051. C. Labbé and D. Labbé. Duplicate and fake publications in the scientific literature: how many SCIgen papers in computer science? HAL Id: hal-00641906, HAL archives-ouvertes.fr, July 2012. 9

1052. T. Labiner. A big decision: Lease or buy? *Computers and Automation*, 6(10):6–8, Oct. 1957. 101

1053. W. Labov. The boundaries of words and their meaning. In C.-J. N. Bailey and R. W. Shuy, editors, *New ways of analyzing variation of English*, pages 340–373. Georgetown Press, 1973. 41

1054. W. Labov. *Principles of Linguistic Change, volume 3: Cognitive and Cultural Factors*. Wiley-Blackwell, 2010. 189

1055. E. Labro and L. Stice-Lawrence. Updating accounting systems: Longitudinal evidence from the health care sector. *Management Science*, ???(???):???, Apr. 2019. 88

1056. J. C. Lagarias. *The Kepler Conjecture: The Hales-Ferguson Proof by Thomas C. Hales Samuel P. Ferguson*. Springer, 2010. 144

1057. K. Laitinen. *Natural naming in software development and maintenance*. PhD thesis, University of Oulu, Finland, Oct. 1995. 189

1058. G. Lakoff and M. Johnson. *Metaphors We Live By*. The University of Chicago Press, 1980. 44, 102

1059. A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31(1):66–104, Apr. 1999. 90

1060. B. L. Lambert, K.-Y. Chang, and P. Gupta. Effects of frequency and similarity neighborhoods on pharmacists' visual perception of drug names. *Social Science and Medicine*, 57(10):1939–1955, Nov. 2003. 191

1061. R. Lämmel, E. Pek, and J. Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC'11, pages 1317–1324, Mar. 2011. 203, 204

1062. B. W. Lampson. A critique of "an exploratory investigation of programmer performance under on-line and off-line conditions". *IEEE Transactions on Human Factors in Electronics*, 8(1):33–48, Mar. 1967. 8

1063. T. K. Landauer. How much do people remember? Some estimates of the quantity of learned information in long-term memory. *Cognitive Science*, 10:477–493, 1986. 55

1064. R. M. Landers, J. B. Rebitzer, and L. J. Taylor. Rat race reduce: Adverse selection in the determination of work hours in law firms. *The American Economic Review*, 86(3):329–348, June 1996. 74

1065. D. Landman, A. Serebrenik, E. Bouwers, and J. J. Vinju. Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions. *Journal of Software: Evolution and Process*, 28(7):589–618, July 2016. 159, 175, 176, 181, 187, 194

1066. D. Landy, D. Brookes, and R. Smout. Abstract numeric relations and the visual structure of algebra. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 40(5):1404–1418, Sept. 2014. 25

1067. D. Landy, A. Charlesworth, and E. Ottmar. Categories of large numbers in line estimation. *Cognitive Science*, 41(2):326–353, Mar. 2017. 46

1068. D. Landy and R. L. Goldstone. Proximity and precedence in arithmetic. *The Quarterly Journal of Experimental Psychology*, 63(10):1953–1968, Oct. 2010. 210

1069. D. Landy, B. Guay, and T. Marghetis. Bias and ignorance in demographic perception. *Psychonomic Bulletin & Review*, 25(5):1606–1618, Oct. 2018. 49

1070. E. J. Langer. The illusion of control. *Journal of Personality and Social Psychologs*, 32(2):311–328, 1975. 53

1071. R. N. Langlois. External economies and economic progress: The case of the microcomputer industry. *The Business History Review*, 66(1):1–50, 1992. 90

1072. LANL. LANL failure data. http://institute.lanl.gov/data/lanldata.shtml, 2006. 163

1073. L. Lapointe and S. Rivard. A multilevel model of resistance to information technology implementation. *MIS Quarterly*, 29(3):461–492, Sept. 2005. 116

1074. I. Larkin. The cost of high-powered incentives: Employee gaming in enterprise software sales. Technical Report 13-073, Harvard Business School, Feb. 2013. 84

1075. C. Larman and V. R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(6):47–56, June 2003. 127

1076. J. Larres. Performance variance evaluation on Mozilla Firefox. Thesis (m.s.), Victoria University of Wellington, May 2012. 368, 369

1077. R. H. Larson, J. K. Salmon, R. O. Dror, M. M. Deneroff, C. Young, J. Grossman, Y. Shan, J. L. Klepeis, and D. E. Shaw. High-throughput pairwise point interactions in Anton, a specialized machine for molecular dynamics simulation. In *IEEE 14th International Symposium on High Performance Computer Architecture*, HPCA 2008, pages 331–342, Feb. 2008. 108

1078. B. Latané and J. M. Darley. Bystander "apathy". *American Scientist*, 57(2):244–269, June-Sept. 1969. 76

1079. B. Latané, K. Williams, and S. Harkins. Many hands make light the work: The causes and consequences of social loafing. *Journal of Personality and Social Psychology*, 37(6):822–832, 1979. 76

1080. R. Latorre. Effects of developer experience on learning and applying unit test-driven development. *IEEE Transactions on Software Engineering*, 40(4):381–395, Apr. 2014. 35, 36

1081. P. R. Laughlin. *Group Problem Solving*. Princeton University Press, Apr. 2015. 75

1082. J. Laukemann, J. Hammer, J. Hofmann, G. Hager, and G. Wellein. Automated instruction stream throughput prediction for Intel and AMD microarchitectures. In *IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, PMBS 2018, pages 121–131, Nov. 2018. 200

1083. E. Laukkanen, M. Paasivaara, J. Itkonen, C. Lassenius, and T. Arvonen. Towards continuous delivery by reducing the feature freeze period: A case study. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP'17, pages 23–32, May 2017. 136

1084. S. Laumer, C. Maier, A. Eckhardt, and T. Weitzel. Work routines as an object of resistance during information systems implementations: theoretical foundation and empirical evidence. *European Journal of Information Systems*, 25(4):317–343, July 2016. 116

1085. L. Lauterbach. Development of N-version software samples for an experiment in software fault tolerance. NASA Contractor Report 178363, Software Research and Development Center for Digital Systems Research, Sept. 1987. 126, 157

1086. C. W. Lazar. Lease/buy decisions for computer acquisition under conditions of uncertain technological change. In *Proceedings–1968 ACM National Conference*, pages 685–690, Jan. 1968. 101

1087. E. Lazear and M. Gibbs. *Personnel Economics for Managers*. John Wiley & Sons, Inc, second edition, 2007. 68, 104, 137

1088. C. Lebiere. *The Dynamics of Cognition: An ACT-R Model of Cognitive Arithmetic*. PhD thesis, Carnegie Mellon University, Nov. 1998. 46

1089. P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4):1–22, Aug. 2007. 161

1090. A. L. Lederer and J. Prasad. Causes of inaccurate software development cost estimates. *Journal of Systems and Software*, 31(2):125–134, Nov. 1995. 121

1091. B. C. Lee and D. M. Brooks. Regression modeling strategies for microarchitecture performance and power prediction. Technical Report TR-08-06, Division of Engineering and Applied Sciences, Harvard University, Mar. 2006. 320, 359

1092. D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu. Adaptive-latency DRAM: Optimizing DRAM timing for the common-case. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, HPCA'15, pages 489–501, Feb. 2015. 367

1093. M. D. Lee, K. A. Gluck, and M. M. Walsh. Understanding the complexity of simple decisions: Modeling multiple behaviors and switching strategies. *Decision*, 6(4):335–368, Oct. 2019. 49

1094. G. Leech, P. Rayson, and A. Wilson. *Word Frequencies in Written and Spoken English*. Pearson Education, 2001. 44, 152, 195

1095. L. Lefebvre and N. J. Boogert. Avian social learning. In M. D. Breed and J. Moore, editors, *Encyclopedia of Animal Behavior: volume 1*, pages 124–130. Oxford: Academic Press, July 2010. 71

1096. J.-A. LeFevre and J. Liu. The role of experience in numerical skill: Multiplication performance in adults from Canada and China. *Mathematical Cognition*, 3(1):31–62, 1997. 48

1097. G. E. Legge, T. A. Hooven, T. S. Klitz, J. S. Mansfield, and B. S. Tjan. Mr. Chips 2002: New insights from an ideal-observer model of reading. *Vision Research*, 42(18):2219–2234, Aug. 2002. 26

1098. C. Leggett. The Ford Pinto case: The valuation of life as it applies to the negligence-efficiency argument. `https://users.wfu.edu/palmitar/Law&Valuation/Papers/1999/Leggett-pinto.html`, 1999. 151

1099. D. R. Lehman, R. O. Lempert, and R. E. Nisbett. The effects of graduate training on reasoning. *American Psychologist*, 43(6):431–442, 1988. 38

1100. L. Lehmann, K. Aoki, and M. W. Feldman. On the number of independent cultural traits carried by individuals and populations. *Philosophical Transactions of The Royal Society B*, 366(1563):424–435, Feb. 2011. 71, 99

1101. P. Lemaire and M. Fayol. When plausibility judgments supersede fact retrieval: The example of the odd-even effect on product verification. *Memory & Cognition*, 23(1):34–48, Feb. 1995. 47

1102. P. Lennie. The cost of cortical computation. *Current Biology*, 13(6):493–497, Mar. 2003. 54

1103. F. Lequiller, N. Ahmad, S. Varjonen, W. Cave, and K.-H. Ahn. Report of the OECD task force on software measurement in the national accounts. STD/NA (2002)2, Organisation for Economic Co-operation and Development, Sept. 2002. 78

1104. K. Lerman, X. Yan, and X.-Z. Wu. The "majority illusion" in social networks. *PLoS ONE*, 11(2):e0147617, Feb. 2016. 96

1105. K. Letrud and S. Hernes. Affirmative citation bias in scientific myth debunking: A three-in-one case study. *PLoS ONE*, 14(9):e0222213, Sept. 2019. 7

1106. D. E. Levari, D. T. Gilbert, T. D. Wilson, B. Sievers, D. M. Amodio, and T. Wheatley. Prevalence-induced concept change in human judgment. *Science*, 360(6396):1465–1467, June 2018. 49, 50

1107. B. W. Leverett, R. G. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz, and W. A. Wulf. An overview of the production-quality compiler-compiler project. Technical Report CMU-CS-79-105, Carnegie Mellon University, Feb. 1979. 174

1108. P. Lewicki, T. Hill, and E. Bizot. Acquisition of procedural knowledge about a pattern of stimuli that cannot be articulated. *Cognitive Psychology*, 20(1):24–37, Jan. 1988. 188

1109. A. C. Lewis. A study of idea generation over time. Thesis (m.s.), Georgia Institute of Technology, June 1972. 76

1110. G. Lewis and P. Bajari. Incentives and adaptation: Evidence from highway procurement in Minnesota. Working Paper 17647, National Bureau of Economic Research, USA, Dec. 2011. 125

1111. J. R. Lewis. Evaluation of procedures for adjusting problem-discovery rates estimated from small samples. *International Journal of Human-Computer Interaction*, 13(4):445–479, Dec. 2001. 166

1112. K. Li, E. Yan, and Y. Feng. How is R cited in research outputs? Structure, impacts, and citation standard. *Journal of Informetrics*, 11(4):989–1002, Nov. 2017. 10

1113. L. Li, T. F. Bissyandé, and J. Klein. MoonlightBox: Mining Android API histories for uncovering release-time inconsistencies. In *IEEE 29th International Symposium on Software Reliability Engineering*, ISSRE'18, pages 212–223, Oct. 2018. 80

1114. L. Li, T. F. Bissyandé, Y. L. Traon, and J. Klein. Accessing inaccessible android APIs: An empirical study. In *International Conference on Software Maintenance and Evolution*, ICSME 2016, pages 411–422, Oct. 2016. 113

1115. Q. Li and H. Pham. A testing-coverage software reliability model considering fault removal efficiency and error generation. *PLoS ONE*, 12(7):e0181524, July 2017. 154

1116. X. Li. *Soft Error Modeling and Analysis for Microprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, May 2008. 162

1117. X. Li, L. Molleman, and D. van Dolder. Conditional punishment: descriptive social norms drive negative reciprocity. Working Paper 3571220, ???, May 2020. 75

1118. Y. Li. Empirical study of Python call graph. In *34th IEEE/ACM International Conference on Automated Software Engineering*, ASE'19, pages 1274–1276, Nov. 2019. 354

1119. Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, Mar. 2006. 78

1120. Y. Liang, Y. Zhang, A. Sivasubramaniam, R. K. Sahoo, J. Moreira, and M. Gupta. Filtering failure logs for a BlueGene/L prototype. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN'05, pages 476–485, June 2005. 381

1121. S. Lichtenstein and B. Fishhoff. Do those who know more also know more about how much they know? *Organizational Behavior and Human Performance*, 20:159–183, 1977. 53

1122. S. Lichtenstein, P. Slovic, B. Fischhoff, M. Layman, and B. Combs. Judged frequency of lethal events. *Journal of Experimental Psychology: Human Learning and Memory*, 4(6):551–578, Nov. 1978. 148

1123. Y. Lichtenstein and A. McDonnell. Pricing software development services. In *European Conference on Information Systems*, ECIS 2003, 2003. 120

1124. C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'15, pages 65–76, June 2015. 151

1125. G. A. Liebchen and M. Shepperd. Data sets and data quality in software engineering. In *Proceedings of the 4th international workshop on Predictor Models in Software Engineering*, PROMISE'08, pages 39–44, May 2008. 373

1126. J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, ICSE'10, pages 105–114, May 2010. 193

1127. J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, ESEC/FSE'13, pages 81–91, Aug. 2013. 168

1128. F. Lieder, T. L. Griffiths, Q. J. M. Huys, and N. D. Goodman. The anchoring bias reflects rational use of cognitive resources. *Psychonomic Bulletin & Review*, 25(1):322–349, Feb. 2018. 22

1129. J. H. Lienhard. *How Invention Begins: Echoes of Old Voices in the Rise of New Machines*. Oxford University Press, 2006. 92

1130. J. S. Light. When computers were women. *Technology and Culture*, 40(3):455–483, July 1999. 104

1131. S. L. Lim. *Social Networks and Collaborative Filtering for Large-Scale Requirements Elicitation*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Aug. 2010. 131, 132, 140

1132. S. L. Lim, P. J. Bentley, N. Kanakam, F. Ishikawa, and S. Honiden. Investigating country differences in mobile app user behavior and challenges for software engineering. *IEEE Transactions on Software Engineering*, 41(1):40–64, Jan. 2015. 101

1133. B. Lin, L. Ponzanelli, A. Mocci, G. Bavota, and M. Lanza. On the uniqueness of code redundancies. In *IEEE/ACM 25th International Conference on Program Comprehension*, ICPC 2017, pages 121–131, May 2017. 195

1134. B. Lin, F. Zampetti, G. Bavota, M. Di Penta, M. Lanza, and R. Oliveto. Sentiment analysis for software engineering: How far can we go? In *Proceedings of the 40th International Conference on Software Engineering*, ICSE'18, pages 94–104, May-June 2018. 346

1135. D.-Y. Lin and I. Neamtiu. Collateral evolution of applications and databases. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution and Software Evolution workshops*, IWPSE-Evol'09, pages 31–40, Aug. 2009. 198

1136. L. C. H. Lin and N. Shen. GPL-3.0 in the Chinese intellectual property court in Beijing. *International Free and Open Source Software Law Review*, 10(1):1–7, 2018. 66

1137. M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. API change and fault proneness: A threat to the success of Android apps. In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, ESEC/FSE'13, pages 477–487, Aug. 2013. 151

1138. K. R. Linberg. Software developer perceptions about software project failure: a case study. *The Journal of Systems and Software*, 49(2–3):177–192, Dec. 1999. 116

1139. K. Lind and R. Heldal. A practical approach to size estimation of embedded software components. *IEEE Transactions on Software Engineering*, 38(5):993–1007, Sept.-Oct. 2012. 126, 127

1140. R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36(4):119–150, Dec. 2004. 357

1141. T. Little. Schedule estimation and uncertainty surrounding the cone of uncertainty. *IEEE Software*, 23(3):48–54, May 2006. 130, 131

1142. D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers*, chapter 6, pages 80–98. Ablex Publishing Corporation, 1986. 182

1143. S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Analysis of the Linux kernel evolution using code clone coverage. In *Fourth International Workshop on Mining Software Repositories*, MSR'07, pages 22–25, May 2007. 207

1144. G. D. Logan. Shapes of reaction-time distributions and shapes of learning curves: A test of the instance theory of automaticity. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 18(5):883–914, 1992. 34

1145. C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. DéjàVu: A map of code duplicates on GitHub. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA'17, page 84, Oct. 2017. 196

1146. C. V. Lopes and J. Ossher. How scale affects structure in Java programs. In *eprint arXiv:cs.SE/1508.00628*, Aug. 2015. 176

1147. I. Lorge and H. Solomon. Two models of group behavior in the solution of eureka-type problems. *Psychometrika*, 20(2):139–148, June 1955. 76

1148. M. Lorko, M. Servátka, and L. Zhang. Anchoring in project duration estimation. *Journal of Economic Behavior & Organization*, 162:49–65, June 2019. 37

1149. D. D. Loschelder, M. Friese, M. Schaerer, and A. D. Galinsky. The too-much-precision effect: When and why precise anchors backfire with experts. *Psychological Science*, 27(12):1573–1587, Oct. 2016. 119

1150. R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski. Evolution of the Linux kernel variability model. In *Proceedings of the 14th International Conference on Software Product Lines: going beyond*, SPLC'10, pages 136–150, Sept. 2010. 328, 329, 330

1151. P. Louridas, D. Spinellis, and V. Vlachos. Power laws in software. *ACM Transactions on Software Engineering and Methodology*, 18(1):1–26, Sept. 2008. 315

1152. L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of Linux file system evolution. In *11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 31–44, Feb. 2013. 181, 215

1153. J. D. Lucente. *On the Viability of Quantitative Assessment Methods in Software Engineering and Software Services*. PhD thesis, School of Engineering and Computer Science, University of Denver, Jan. 2015. 151, 152

1154. L. Lucia. *Ranking-Based Approaches for Localizing Faults*. PhD thesis, Singapore Management University, June 2014. 160, 161

1155. L. Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, 26(2):172–219, Feb. 2014. 159

1156. K. M. Lui and K. C. C. Chan. Pair programming productivity: Novice-novice vs. expert-expert. *International Journal of Human-Computer Studies*, 64(9):915–925, Sept. 2006. 34, 35

1157. P. Lukowicz, E. A. Heinz, L. Prechelt, and W. F. Tichy. Experimental evaluation in computer science: A quantitative study. Technical Report 17/94, University of Karlsruhe, Germany, Aug. 1994. 6

1158. A. Lundqvist and D. Rodic. GNU/Linux distribution timeline. website, Oct. 2012. http://futurist.se/gldt. 95

1159. M. I. Lunesu. *Process Software Simulation Model of Lean-Kanban Approach*. PhD thesis, Department of Electrical and Electronic Engineering, University of Cagliari, Apr. 2013. 339

1160. Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 467–478, June 2014. 162

1161. A. K. Luria. Towards the problem of the historical nature of psychological processes. *International Journal of Psychology*, 6(4):259–272, 1971. 19, 41

1162. A. R. Luria. *The Mind of a Mnemonist*. Penguin Education, 1975. 32

1163. B. Luthiger and C. Jungwirth. Pervasive fun. *First Monday*, 12(1), Jan. 2007. 302

1164. W. J. Lynn III. A new approach for delivering information technology capabilities in the department of defense. Report to congress, Office of the Secretary of Defense, Nov. 2010. 127

1165. W. Ma, L. Chen, X. Zhang, Y. Zhou, and B. Xu. How do developers fix cross-project correlated bugs? A case study on the GitHub scientific Python ecosystem. In *IEEE/ACM 39th International Conference on Software Engineering*, ICSE'17, pages 381–392, May 2017. 148

1166. W. Ma, J.-C. S. Liu, and A. Forin. Design and testing of a cpu emulator. Technical Report MSR-TR-2009-155, Microsoft Research, Aug. 2009. 161

1167. F. MacCrory, V. Choudhary, and A. Pinsonneault. Designing promotion ladders to mitigate turnover of IT professionals. *Information Systems Research*, 27(3):648–660, Sept. 2016. 67

1168. N. Macdonald. Computing services survey. *Computers and Automation*, 7(7):9–12, July 1958. 101

1169. N. Macdonald. *Computer Census 1962-74*. Computers and People, 1974. 101

1170. J. N. MacGregor. Short-term memory capacity: Limitation or optimization? *Psychological Review*, 94(1):107–108, Jan. 1987. 32

1171. A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, ESEC/FSE'13, pages 224–234, Aug. 2013. 343, 344, 345

1172. C. E. Mackenzie. *Coded Character Sets, History and Development*. Addison–Wesley, 1980. 102

1173. D. MacKenzie. Computer-related accidental death: an empirical exploration. *Science and Public Policy*, 21(4):233–248, Aug. 1994. 149

1174. I. S. MacKenzie. Fitts' law as a research and design tool in human-computer interaction. *International Journal of Human-Computer Interaction*, 7(1):91–139, Mar. 1992. 56

1175. R. J. Madachy. *Software Process Dynamics*. John Wiley & Sons, Inc, 2008. 124, 352

1176. A. Maddison. Business cycles, long waves and phases of capital development. In A. Maddison, editor, *Dynamic Forces in Capitalist Development: A Long-run Comparative View*, chapter 4, page ??? Oxford University Press, Oct. 1991. 5

1177. W. T. Maddox and C. J. Bohil. Costs and benefits in perceptual categorization. *Memory & Cognition*, 28:597–615, 2000. 38

1178. M. Magidin and E. Viso. On the experiments in algorithm dynamics. Technical Report UAMR0853, Universidad Autónoma Metropolitana-Iztapalapa, México, Oct. 1976. 194

1179. T. Maillart, M. Zhao, J. Grossklags, and J. Chuang. Given enough eyeballs, all bugs are shallow? Revisiting Eric Raymond with bug bounty programs. *Journal of Cybersecurity*, 3(2):81–90, June 2017. 105

1180. S. Majd and R. S. Pindyck. Time to build, option value, and investment decisions. *Journal of Financial Economics*, 18(1):7–27, Mar. 1987. 58

1181. V. Makarov. 2016 Export of Russian software development industry. Annual Survey 13-th, RUSSOFT Association, Aug. 2016. 58, 109

1182. V. N. Makarow. SPEC benchmark page. http://vmakarov.fedorapeople.org/spec/index.html, July 2014. 369, 370

1183. B. A. Malloy and J. F. Power. Quantifying the transition from Python 2 to 3: An empirical study of Python applications. In *International Symposium on Empirical Software Engineering and Measurement*, ESEM'17, pages 314–323, Dec. 2017. 197

1184. S. Mandal, R. Gandhi, and H. Siy. Modular norm models: practical representation and analysis of contractual rights and obligations. *Requirements Engineering*, 25(3):383–412, Sept. 2020. 121

1185. M. Mangel and F. J. Samaniego. Abraham Wald's work on aircraft survivability. *Journal of the American Statistical Association*, 79(386):259–267, June 1984. 251

1186. Manpower. Computers in offices. Studies No. 4, Manpower Research Unit, Ministry of Labour, G.B., 1965. 88

1187. M. M. Mantei and T. J. Teorey. Cost/benefit analysis for incorporating human factors in the software lifecycle. *Communications of the ACM*, 31(4):428–438, Apr. 1988. 126

1188. M. V. Mäntylä and J. Itkonen. How are software defects found? The role of implicit defect detection, individual responsibility, documents, and knowledge. *Information and Software Technology*, 56(12):1597–1612, Dec. 2014. 169

1189. A. Marathe, Y. Zhang, G. Blanks, N. Kumbhare, G. Abdulla, and B. Rountree. An empirical survey of performance and energy efficiency variation on Intel processors. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*, E2SC'17, pages 1–8, Nov. 2017. 365

1190. A. Marchand. The power of an installed base to combat lifecycle decline: The case of video games. *International Journal of Research in Marketing*, 33(1):140–154, Mar. 2016. 82

1191. M. Marcozzi, Q. Tang, A. F. Donaldson, and C. Cadar. Compiler fuzzing: How much does it matter? *ACM Transactions on Programming Languages and Systems*, 3:155, Oct. 2019. 166, 167

1192. J. N. Marewski and L. J. Schooler. Cognitive niches: An ecological model of strategy selection. *Psychological Review*, 118(3):292–437, July 2011. 51

1193. B. H. Margolin, R. P. Parmelee, and M. Schatzoff. Analysis of free-storage algorithms. *IBM Systems Journal*, 10(4):283–304, 1971. 198

1194. P. Marinescu, P. Hosek, and C. Cadar. COVRIG: A framework for the analysis of code, test, and coverage evolution in real software. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA'14, pages 93–104, July 2014. 168

1195. F. Marotta-Wurgler. What's in a standard form contract? An empirical analysis of software license agreements. *Journal of Empirical Legal Studies*, 7(4):677–713, Dec. 2007. 120

1196. F. Marotta-Wurgler. Will increased disclosure help? Evaluating the recommendations of the ALI's "principles of the law of software contracts". *University of Chicago Law Review*, 78(1), 2011. 66

1197. D. Martin. *An Empirical Analysis of GNU Make Feature Use in Open Source Projects*. PhD thesis, School of Computing, Queen's University, Ontario, Apr. 2017. 193

1198. F. Martineau. PNFG: A framework for computer game narrative analysis. Thesis (m.s.), School of Computer Science, McGill University, June 2006. 180

1199. A. G. Martínez. *Chaos Monkeys: Inside The Silicon Valley Money Machine*. Ebury Press, 2016. 89

1200. M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. In *eprint arXiv:cs.SE/1311.3414v1*, Nov. 2013. 188

1201. E. Masanet, A. Shehabi, J. Liang, L. Ramakrishnan, X. Ma, V. Hendrix, B. Walker, and P. Mantha. The energy efficiency potential of cloud-based software: A U.S. case study. LBNL Paper LBNL-6298E, Lawrence Berkeley National Laboratory, June 2013. 91

1202. F. Massacci, S. Neuhaus, and V. H. Nguyen. After-life vulnerabilities: A study on Firefox evolution, its vulnerabilities, and fixes. In *Proceedings of the Third international conference on Engineering secure software and systems*, ESSoS'11, pages 195–208, Feb. 2011. 156, 158, 159

1203. R. C. Masse, C. Liu, Y. Li, L. Mai, and G. Cao. Energy storage through intercalation reactions: electrodes for rechargeable batteries. *National Science Review*, 4(1):26–53, 2017. 364

1204. J. Matejka and G. Fitzmaurice. Same stats, different graphs: Generating datasets with varied appearance and identical statistics through simulated annealing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI'17, pages 1290–1294, May 2017. 382

1205. B. G. Mateus and M. Martinez. On the adoption, usage and evolution of Kotlin features on Android development. In *eprint arXiv:cs.CS/1907.09003*, July 2019. 109

1206. F. Mathy, M. Chekaf, and N. Cowan. Simple and complex working memory tasks allow similar benefits of information compression. *Journal of Cognition*, 1(31):1–12, May 2018. 30, 31

1207. E. Matias, I. S. MacKenzie, and W. Buxton. One-handed touch-typing on a QWERTY keyboard. *International Journal of Human-Computer Interaction*, 11:1–27, 1996. 21

1208. C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'12, pages 683–702, Oct. 2012. 201

1209. D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig. Understanding the use of lambda expressions in Java. In *Proceedings of the ACM on Programming Languages*, OOPSLA'17, page 85, Oct. 2017. 197

1210. A. Mazouz. *An Empirical Study of Program Performance of OpenMP Applications on Multicore Platforms*. PhD thesis, Université de Versailles-Saint Quentin en Yvelines, Dec. 2013. 368, 369

1211. M. Mazzucato. Risk, variety and volatility in the PC industry: *New* economy or *Early* life-cycle? In *NY Federal Reserve Bank conference on "Productivity Growth: A New Era?"*, Nov. 2001. 96

1212. D. F. McAllister and M. A. Vouk. Experiments in fault tolerant software reliability. Technical Report 5 – NAG-1-667, North Carolina State University, Apr. 1989. 126, 171

1213. T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec. 1976. 194

1214. J. C. McCallum. Historical cost of computer memory and storage. http://www.jcmit.com, July 2016. 1

1215. S. McCloud. *Understanding Comics*. HarperPerennial, 1993. 224

1216. S. McConnell. *Code Complete*. Microsoft Press, 1993. 179

1217. M. H. McCormack. *The Terrible Truth about Lawyers*. Beech Tree books, William Morrow, 1987. 120

1218. M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, 33(4):125–180, June 2001. 357

1219. R. R. McCrae and P. T. Costa, Jr. Reinterpreting the Myers-Briggs type indicator from the perspective of the five-factor model of personality. *Journal of Personality*, 57(1):17–40, Mar. 1989. 50

1220. B. D. McCullough. Microsoft Excel's 'Not The Wichmann-Hill' random number generator. *Computational Statistics & Data Analysis*, 52:4587–4593, 2008. 161

1221. B. D. McCullough and D. A. Heiser. On the accuracy of statistical procedures in Microsoft Excel 2007. *Computational Statistics & Data Analysis*, 52:4570–4578, 2008. 14

1222. J. Mcdonald. The impact of project planning team experience on software project cost estimates. *Empirical Software Engineering*, 10(2):219–234, Apr. 2005. 121, 122

1223. R. McElreath and R. Boyd. *Mathematical Models of Social Evolution: A Guide for the Perplexed*. The University of Chicago Press, 2008. 71, 96

1224. D. McFadden. Rationality for economists? *Journal of Risk and Uncertainty*, 19:73–105, 1999. 51

1225. R. W. McGee. *Accounting for Software in the United States*. PhD thesis, School of Industrial and Business Studies, University of Warwick, Apr. 1986. 78

1226. B. McGonigle, M. Chalmers, and A. Dickinson. Concurrent disjoint and reciprocal classification by Cebus apella in seriation tasks: evidence for hierarchical organization. *Animal Cognition*, 6(3):185–197, Sept. 2003. 38

1227. S. McJohn. The GPL meets the UCC: Does free software come with a warranty of no infringement? *Journal of High Technology Law*, XV(1):1–62, 2014. 66

1228. K. B. McKeithen, J. S. Reitman, H. H. Ruster, and S. C. Hirtle. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13(3):307–325, July 1981. 37, 38

1229. J. McManus and T. Wood-Harper. Understanding the sources of information systems project failure: A study in IS project failure. *Management Services*, 51(3):38–43, Aug. 2007. 118

1230. D. S. McNamara and J. Magliano. Toward a comprehensive model of comprehension. In B. Ross, editor, *The Psychology of Learning and Motivation, Vol. 51*, chapter 9, pages 297–384. Academic Press, Nov. 2009. 185

1231. T. P. McNamara, J. K. Hardy, and S. C. Hirtle. Subjective hierarchies in spatial memory. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 15(2):211–227, 1989. 28

1232. J. McNerney, J. D. Farmer, S. Redner, and J. E. Trancik. Role of design complexity in technology improvement. *Proceedings of the National Academy of Sciences*, 108(22):9008–9013, May 2011. 72

1233. D. L. McNicol. Influences on the timing and frequency of cancellations and truncations of major defense acquisition programs. IDA Paper P-8280, Institute for Defense Analyses, Mar. 2017. 118

1234. I. McPhee. Customs' cargo management re-engineering project: Australian customs service. Audit Report No.24 2006-07, Australian National Audit Office, Aug. 2007. 118

1235. J. H. McWhorter. The world's simplest grammars are creole grammars. *Linguistic Typology*, 5(2-3):125–166, 2001. 197

1236. A. D. Meacham. *Data Processing Equipment Encyclopedia: Electronic Devices*, volume 2. Gille Associates, Inc., 1962. 108, 361

1237. F. Mechner. Probability relations within response sequences under ratio reinforcement. *Journal of the Experimental Analysis of Behavior*, 1(2):109–121, Apr. 1958. 18

1238. F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. A comparison of 10 sampling algorithms for configurable systems. In *eprint arXiv:cs.SE/1602.02052*, Feb. 2016. 136, 164, 251

1239. F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The love/hate relationship with the C preprocessor: An interview study. In *Proceedings of the 29th. European Conference on Object-Oriented Programming*, ECOOP'15, pages 495–518, July 2015. 179

1240. M. Meeks. German comments in LibreOffice. website, Apr. 2017. https://people.gnome.org/~michael/data/2015-08-01-5.5-data.ods. 103

1241. M. Meeter, J. M. J. Murre, and S. M. J. Janssen. Remembering the news: Modeling retention data from a study with 14,000 participants. *Memory & Cognition*, 33(5):793–810, July 2004. 33, 34

1242. M. Mehrara and T. Austin. Exploiting selective placement for low-cost memory protection. *ACM Transactions on Architecture and Code Optimization*, 5(3):1–24, Nov. 2008. 162

1243. L. K. Melhus and R. E. Jensen. Measurement bias from address aliasing. In *Eleventh International Workshop on Automatic Performance Tuning*, iWAPT 2016, pages 1506–1515, May 2016. 366

1244. R. Meloca, G. Pinto, L. Baiser, M. Mattos, I. Polato, I. S. Wiese, and D. M. German. Understanding the usage, impact, and adoption of non-OSI approved licenses. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR'18, pages 270–280, May 2018. 65, 66

1245. M. Mencher. *Get in the Game! Careers in the Game Industry*. New Riders Publishing, Oct. 2002. 104

1246. D. Mendez, B. Baudry, and M. Monperrus. Empirical evidence of large-scale diversity in API usage of object-oriented software. In *International Conference on Source Code Analysis and Manipulation*, SCAM'13, pages 43–52, Apr. 2013. 203, 204

1247. H. Mengistu, J. Huizinga, J.-B. Mouret, and J. Clune. The evolutionary origins of hierarchy. *PLoS Computational Biology*, 12(6):e1004829, June 2016. 179

1248. H. Mercier and D. Sperber. Why do humans reason? Arguments for an argumentative theory. *Behavioral and Brain Sciences*, 34(2):57–111, Apr. 2011. 41

1249. H. Mercier and D. Sperber. Why do humans reason? Arguments for an argumentative theory. HAL Id: hal-00904097, HAL archives-ouvertes.fr, Nov. 2013. 41

1250. R. C. Merkle. Energy limits to the computational power of the human brain. *Foresight Update*, 6, Aug. 1989. 54

1251. E. W. Merrow, L. McDonnel, and R. W. Argüden. Understanding the outcomes of megaprojects: A quantitative analysis of very large civilian projects. Report R-3560-PSSP, The RAND Corporation, Mar. 1988. 121

1252. S. Mertens and C. Baethge. The virtues of correct citation. *Deutsches Ärzteblatt International*, 108(33):550–552, Apr. 2011. 144

1253. A. Mesoudi. Cultural evolution: A review of theory, findings and controversies. *Evolutionary Biology*, 43(4):481–497, Dec. 2016. 69

1254. A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz. The work life of developers: Activities, switches and perceived productivity. *IEEE Transactions on Software Engineering*, 43(12):1178–1193, Dec. 2017. 133

1255. L. A. Meyerovich and A. Rabkin. How not to survey developers and repositories: Experiences analyzing language adoption. In *Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU'12, pages 7–16, Oct. 2012. 111

1256. X. Mi, Y. Zhang, F. Qian, and X. Wang. An empirical characterization of IFTTT: Ecosystem, usage, and performance. In *Proceedings of the 2017 Internet Measurement Conference*, IMC'17, pages 398–404, Nov. 2017. 175

1257. T. Micceri. The unicorn, the normal curve, and other improbable creatures. *Psychological Bulletin*, 105(1):156–166, Apr. 1989. 249

1258. S. E. Michalak, A. J. DuBois, C. B. Storlie, H. M. Quinn, W. N. Rust, D. H. DuBois, D. G. Modl, A. Manuzzato, and S. P. Blanchard. Assessment of the impact of cosmic-ray-induced neutrons on hardware in the Roadrunner supercomputer. *IEEE Transactions on Device and Materials Reliability*, 12(2):445–454, May 2012. 163

1259. J.-B. Michel, Y. K. Shen, A. P. Aiden, A. Veres, M. K. Gray, The Google Books Team, J. P. Pickett, D. Hoiberg, D. Clancy, P. Norvig, J. Orwant, S. Pinker, M. A. Nowak, and E. L. Aiden. Quantitative analysis of culture using millions of digitized books. *Science*, 14(6014):176–182, Jan. 2011. 110, 381

1260. Microsoft server protocol documentation. website, 2015. http://www.microsoft.com. 77, 114, 161, 218

1261. S. Milgram. *Obedience to Authority*. McGraw-Hill, 1974. 52

1262. S. Milgram, L. Bickman, and L. Berkowitz. Note on the drawing power of crowds of different size. *Journal of Personality and Social Psychologs*, 13(2):79–82, 1969. 71

1263. A. Mili, S. F. Chmiel, R. Gottumukkala, and L. Zhang. Managing software reuse economics: An integrated ROI-based model. *Annals of Software Engineering*, 11(1):175–218, Nov. 2001. 77

1264. K. Milis. *Success factors for ICT projects: Empirical research, utilising qualitative and quantitative approaches (incl. Bayesian networks, Probabilistic feature models)*. PhD thesis, Toegepaste Economische Wetenschappen, Limburgs Universitair Centrum, Dec. 2002. 116

1265. D. M. Miller. Application of halstead's timing model to predict the compilation time of Ada compilers. Thesis (m.s.), School of Engineering of the Air Force Institute of Technology, USA, Dec. 1986. 176

1266. D. R. Miller. Exponential order statistic models of software reliability growth. NASA Contractor Report 3909, NASA Langley Research Center, July 1985. 99, 148, 154

1267. G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63(2):81–97, Mar. 1956. 28, 358

1268. G. A. Miller and S. Isard. Free recall of self-embedded English sentences. *Information and Control*, 7:292–303, 1964. 30

1269. L. A. Miller. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal*, 29(2):184–215, 1981. 193

1270. S. J. Miller and M. J. Nigrini. Order statistics and Benford's law. *International Journal of Mathematics and Mathematical Sciences*, 2008, 2008. 382

1271. W. R. Miller and M. Sanchez-Craig. How to have a high success rate in treatment: advice for evaluators of alcoholism programs. *Addiction*, 91(6):779–785, Apr. 1996. 355

1272. S. MINAKAWA, T. HIRATA, K. MASAME, H. OKADA, and K. MARUYAMA. A psychological analysis on the meaning of "reliance". *Tohoku Psychologica Folia*, 46(1-4):111–117, Apr. 1987. 148

1273. C. H. Mireles. *Marketing Modeling for New Products*. PhD thesis, Erasmus University, Rotterdam, June 2010. 83

1274. MISRA. *MISRA-C:2004 Guidelines for the Use of the C Language in Vehicle Based Software*. Motor Industry Research Association, 2004. 149, 179, 199

1275. MISRA. *MISRA-C++:2008 Guidelines for the Use of the C++ Language in Critical Systems*. Motor Industry Research Association, June 2008. 149

1276. K. E. Mitchell. The copyleft bust up: loopholes, licenses, and realpolitik in open source. blog: /dev/lawyer blog, Nov. 2018. https://writing.kemitchell.com/2018/11/04/Copyleft-Bust-Up.html. 64

1277. R. K. Mitchell, B. R. Agle, and D. J. Wood. Toward a theory of stakeholder identification and salience: Defining the principle of who and what really counts. *The Academy of Management Review*, 22(4):853–886, Oct. 1997. 132

1278. K. Mitropoulou. *Performance Optimizations for Compiler-based Error Detection*. PhD thesis, School of Informatics, University of Edinburgh, Oct. 2014. 163

1279. S. Mittal. A survey of architectural techniques for managing process variation. *ACM Computing Surveys*, 48(4), May 2016. 363

1280. S. Mittal. A survey of value prediction techniques for leveraging value locality. *Concurrency and Computation: Practice and Experience*, 29(21):e4250, Nov. 2017. 198

1281. S. Mittal. A survey of techniques for dynamic branch prediction. In *eprint arXiv:cs.AR/1804.00261*, Apr. 2018. 198

1282. M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1(2):226–251, 2003. 235

1283. M. Mitzenmacher. Dynamic models for file sizes and double Pareto distributions. *Internet Mathematics*, 1(3):305–333, Apr. 2004. 223, 242

1284. O. Mlouki. On the detection of licenses violations in the Android ecosystem. Thesis (m.s.), Université de Montréal, Apr. 2016. 65

1285. A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance*, ICSM'00, pages 120–130, Oct. 2000. 139

1286. A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2), Apr.-June 2000. 35

1287. S. N. Mohanty. Software cost estimation: Present and future. *Software–Practice and Experience*, 11(2):103–121, Feb. 1981. 123, 124

1288. T. Moher and G. M. Schneider. Methods for improving controlled experimentation in software engineering. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE'81, pages 224–233, Mar. 1981. 357

1289. K. Moløkken-Østvold and K. M. Furulund. The relationship between customer collaboration and software project overruns. In *2007 Agile Conference*, AGILE'07, pages 72–83, Aug. 2007. 307

1290. K. Moløkken-Østvold, M. Jørgensen, S. S. Tanilkan, H. Gallis, A. C. Lien, and S. E. Hove. A survey on software estimation in the norwegian industry. In *Proceedings 10th International Symposium on Software Metrics*, pages 208–219, Sept. 2004. 118, 119

1291. C. Montalvo, D. Peck, and E. Rietveld. A longer lifetime for products: Benefits for consumers and companies. Study IP/A/IMCO/2015-11, Policy Department A: Economic and Scientific Policy, European Parliament, June 2016. 94

1292. G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, Apr. 1965. 4

1293. A. C. Morgan, D. J. Economou, S. F. Way, and A. Clauset. Prestige drives epistemic inequality in the diffusion of scientific ideas. In *eprint arXiv:cs.SI/1805.09966*, Oct. 2018. 70

1294. T. J. H. Morgan, L. E. Rendell, M. Ehn, W. Hoppitt, and K. N. Laland. The evolutionary basis of human social learning. *Proceedings of the Royal Society B: Biological Sciences*, 279(1729):653–662, Jan. 2012. 52

1295. F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6(2):139–143, Apr. 1984. 143

1296. R. J. Morrison, R. E. Nolan, and J. S. Devlin. *Work Measurement in Machine Accounting: Controls, Incentives, Scheduling, and Costing Procedures*. The Ronald Press Company, Dec. 1963. 69

1297. S. P. Morse, B. W. Ravenel, S. Mazor, and W. B. Pohlman. Intel microprocessors: 8008 to 8086. *IEEE Computer*, 13(10):42–60, Oct. 1980. 90

1298. T. Moscibroda and R. Oshman. Resilience of mutual exclusion algorithms to transient memory faults. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC'11, pages 69–78, June 2011. 163

1299. F. Mosteller and C. Youtz. Quantifying probabilistic expressions. *Statistical Science*, 5(1):2–12, Feb. 1990. 149

1300. C. Motta. Analysing the evolution of system requirements – A case study of AUTOSAR at Volvo car group. Thesis (m.s.), Department of Computer Science and Engineering, Gothenburg, June 2016. 101, 102

1301. J. F. Motz. In re microsoft corporation antitrust litigation * sun microsystems, inc. v. microsoft corporation * mdl 1332 * civil no. jfm-02-2739. Opinion, UNITED STATES DISTRICT COURT FOR THE DISTRICT OF MARYLAND, 2002. 106, 167

1302. Y. Moy and A. Wallenburg. Tokeneer: Beyond formal program verification. In *Proceedings of the 5th International Congress on Embedded Real Time Software and Systems, ERTS^2* 2010, May 2010. 164

1303. S. T. Mueller and A. Krawitz. Reconsidering the two-second decay hypothesis in verbal working memory. *Journal of Mathematical Psychology*, 53(1):14–25, Feb. 2009. 29

1304. S. T. Mueller and C. T. Weidemann. Alphabetic letter identification: Effects of perceivability, similarity, and bias. *Acta Psychologica*, 139(1):19–37, Jan. 2012. 21, 191

1305. D. Mulcahy, B. Weeks, and H. S. Bradley. "we have met the enemy… and he is us" Lessons from twenty years of the Kauffman Foundation's investments in venture capital funds and the triumph of hope over experience. Report, Ewing Marion Kauffman Foundation, May 2012. 89

1306. C. W. Mulford and S. Misra. Capitalization of software development costs: Accounting practices in the software industry, 2014 and 2015. Technical report, Georgia Tech College of Management, Jan. 2016. 58, 78, 80

1307. C. W. Mulford and J. Roberts. Capitalization of software development costs: A survey of accounting practices in the software industry. Technical report, Georgia Tech College of Management, May 2006,available. 80

1308. M. M. Müller and A. Höfer. The effect of experience on the test-driven development process. *Empirical Software Engineering*, 12(6):593–615, 2007. 212, 356, 375

1309. J. Mulligan and B. Patrovsky. *Developing Online Games: An Insider's Guide*. New Riders Publishing, 2003. 126

1310. E. Mumford. *Job Satisfaction: A study of computer specialists*. Longman Group Limited, Oct. 1972. 64, 67

1311. A. M. Munñiz Jr. and H. J. Schau. Religiosity in the abandoned Apple Newton brand community. *Journal of Consumer Research*, 31(4):737–747, Mar. 2005. 70

1312. D. Muna, M. Alexander, A. Allen, R. Ashley, D. Asmus, R. Azzollini, M. Bannister, R. Beaton, A. Benson, G. B. Berriman, M. Bilicki, P. Boyce, J. Bridge, J. Cami, E. Cangi, X. Chen, N. Christiny, C. Clark, M. Collins, J. Comparat, N. Cook, D. Croton, I. D. Davids, É. Depagne, J. Donor, L. A. dos Santos, S. Douglas, A. Du, M. Durbin, D. Erb, D. Faes, J. G. Fernández-Trincado, A. Foley, S. Fotopoulou, S. Frimann, P. Frinchaboy, R. Garcia-Dias, A. Gawryszczak, E. George, S. Gonzalez, K. Gordon, N. Gorgone, C. Gosmeyer, K. Grasha, P. Greenfield, R. Grellmann, J. Guillochon, M. Gurwell, M. Haas, A. Hagen, D. Haggard, T. Haines, P. Hall, W. Hellwing, E. C. Herenz, S. Hinton, R. Hlozek, J. Hoffman, D. Holman, B. W. Holwerda, A. Horton, C. Hummels, D. Jacobs, J. J. Jensen, D. Jones, A. Karick, L. Kelley, M. Kenworthy, B. Kitchener, D. Klaes, S. Kohn, P. Konorski, C. Krawczyk, K. Kuehn, T. Kuutma, M. T. Lam, R. Lane, J. Liske, D. Lopez-Camara, K. Mack, S. Mangham, Q. Mao, D. J. E. Marsh, C. Mateu, L. Maurin, J. McCormac, I. Momcheva, H. Monteiro, M. Mueller, R. Munoz, R. Naidu, N. Nelson, C. Nitschelm, C. North, J. Nunez-Iglesias, S. Ogaz, R. Owen, J. Parejko, V. Patrício, J. Pepper, M. Perrin, T. Pickering, J. Piscionere, R. Pogge, R. Poleski, A. Pourtsidou, A. M. Price-Whelan, M. L. Rawls, S. Read, G. Rees, H. Rein, T. Rice, S. Riemer-Sørensen, N. Rusomarov, S. F. Sanchez, M. Santander-García, G. Sarid, W. Schoenell, A. Scholz, R. L. Schuhmann, W. Schuster, P. Scicluna, M. Seidel, L. Shao, P. Sharma, A. Shulevski, D. Shupe, C. Sifón, B. Simmons, M. Sinha, I. Skillen, B. Soergel, T. Spriggs, S. Srinivasan, A. Stevens, O. Streicher, E. Suchyta, J. Tan, O. G. Telford, R. Thomas, C. Tonini, G. Tremblay, S. Tuttle, T. Urrutia, S. Vaughan, M. Verdugo, A. Wagner, J. Walawender, A. Wetzel, K. Willett, P. K. G. Williams, G. Yang, G. Zhu, and A. Zonca. The Astropy problem. In *eprint arXiv:astro-ph.IM/1610.03159*, Oct. 2016. 70, 117

1313. B. B. Murdoch, Jr. The serial position effect of free recall. *Journal of Experimental Psychology*, 64(5):482–488, 1962. 32

1314. E. M. Murphy. In the matter of Knight Capital Americas LLC respondent. Order instituting administrative and cease-and-desist proceedings, pursuant to sections 15(b) and 21c of the securities exchange Act of 1934, making findings, and imposing remedial sanctions and a cease-and-desist order. Administrative Proceeding File No. 3-15570, Securities and Exchange Commission, Oct. 2013. 150

1315. E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE'09, pages 287–297, Apr. 2009. 135

1316. J. M. J. Murre and A. G. Chessa. Power laws from individual differences in learning and forgetting: mathematical analyses. *Psychonomic Bulletin & Review*, 18(3):592–597, June 2011. 243

1317. J. M. J. Murre and J. Dros. Replication and analysis of Ebbinghaus' forgetting curve. *PLoS ONE*, 10(7):e0120644, July 2015. 33

1318. P. Murrell. *R Graphics*. Chapman & Hall/CRC, 1st edition, 2006. 221

1319. M. Muthukrishna, J. Henrich, W. Toyokawa, T. Hamamura, T. Kameda, and S. J. Heine. Overconfidence is universal? Elicitation of genuine overconfidence (EGO) procedure reveals systematic differences across domain, task knowledge, and incentives in four populations. *PLoS ONE*, 13(8):e0202288, Aug. 2018. 53

1320. M. Muthukrishna, B. W. Shulman, V. Vasilescu, and J. Henrich. Sociality influences cultural complexity. *Proceedings of the Royal Society B: Biological Sciences*, 281(1774), Nov. 2013. 73

1321. L. H. Mutuel. Single event effects mitigation techniques report. Final Report DOT/FAA/TC-15/62, U.S. Department of Transportation, Federal Aviation Administration, Feb. 2016. 162

1322. G. J. Myers. A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM*, 21(9):760–768, Sept. 1978. 165

1323. T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. We have it easy, but do we have it right? In *International Symposium on Parallel and Distributed Processing*, IPDPS 2008, pages 1–7, Apr. 2008. 368

1324. T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan. Observer effect and measurement bias in performance analysis. Technical Report CU-CS 1042-08, University of Colorado at Boulder, June 2008. 366

1325. M. Nagappan, R. Robbes, Y. Kamei, É. Tanter, S. McIntosh, A. Mockus, and A. E. Hassan. An empirical study of goto in C code from GitHub repositories. In *Proceedings of the 10th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, ESEC/FSE'15, pages 404–414, Aug.-Sept. 2015. 199

1326. M. Nagappan, T. Zimmermann, and C. Bird. Diversity in software engineering research. In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, ESEC/FSE'13, pages 466–476, Aug. 2013. 251

1327. N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE'10, pages 309–318, Nov. 2010. 311

1328. P. M. Nagel, F. W. Scholz, and J. A. Skrivan. Software reliability: Additional investigations into modeling with replicated experiments. NASA Contractor Report 172378, Boeing Computer Services Company, Space and Military Applications Division, June 1984. 153

1329. P. M. Nagel and J. A. Skrivan. Software reliability: Repetitive run experimentation and modeling. NASA Contractor Report 165836, Boeing Computer Services Company, Space and Military Applications Division, Feb. 1982. 152, 153, 154

1330. T. Nagle, J. Hogan, and J. Zale. *The Strategy and Tactics of Pricing*. Pearson, fifth edition, 2015. 81

1331. J. S. Nairne. The loss of positional certainty in long-term memory. *Psychological Science*, 3(2):199–202, May 1992. 32

1332. J. S. Nairne. Adaptive memory: Evolutionary constraints on remembering. In B. H. Ross, editor, *Psychology of Learning and Motivation, Volume 53*, chapter 1, pages 1–32. Academic Press, June 2010. 27

1333. J. Nandhakumar and D. E. Avison. The fiction of methodological development: a field study of information systems development. *Information Technology & People*, 12(2):176–191, Feb. 1999. 127

1334. S. Nanz and C. A. Furia. A comparative study of programming languages in Rosetta Code. In *eprint arXiv:cs.SE/1409.0252v1*, Aug. 2014. 192

1335. E. Nasseri. *An Empirical Investigation of Inheritance Trends in Java OSS Evolution*. PhD thesis, Department of Information Systems, Computing and Mathematics, Brunel University, June 2009. 204

1336. M. B. Nathanson. Desperately seeking mathematical truth. *Notices of the AMS*, 55(7):773–773, Aug. 2008. 144

1337. P. Naur and B. Randell. Software engineering report on a conference sponsored by the NATO science committee. Technical report, NATO, Jan. 1969. 6, 108

1338. A. D. Navarro and E. Fantino. The sunk cost effect in pigeons and humans. *Journal of the Experimental Analysis of Behavior*, 83(1):1–13, Jan. 2005. 56

1339. D. J. Navarro and A. F. Perfors. Hypothesis generation, sparse categories, and the positive test strategy. *Psychological Review*, 118(1):120–134, Jan. 2011. 23

1340. I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR'05, pages 1–5, May 2005. 204, 205

1341. I. G. Neamtiu. *Practical Dynamic Software Updating*. PhD thesis, University of Maryland, College Park, Aug. 2008. 204

1342. S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In *Proceedings of the 26th European conference on Object-Oriented Programming*, ECOOP'12, pages 79–103, June 2012. 196, 377

1343. D. L. Nelson, C. L. McEvoy, and T. A. Schreiber. The university of South Florida word association, rhyme and word fragment norms. w3.usf.edu/FreeAssociation, 1998. 189

1344. E. A. Nelson. Management handbook for the estimation of computer programming costs. Technical Documentary Report ESD-TDR-67-66, United States Air Force, L. G. Hanscom Field, Bedford, Massachusetts, Oct. 1966. 123

1345. D. A. Nembhard and N. Osothsilp. An empirical comparison of forgetting models. *IEEE Transactions on Engineering Management*, 48(3):283–291, Aug. 2001. 72

1346. R. E. NeSmith II. A study of software maintenance costs of Air Force large scale computer systems. Thesis (m.s.), School of Systems and Logistics, Air Force Institute of Technology, USA, Sept. 1986. 116, 300

1347. D. Nettle. Explaining global patterns of language diversity. *Journal of Anthropological Archaeology*, 17(4):354–374, Dec. 1998. 109

1348. B. New, L. Ferrand, C. Pallier, and M. Brysbaert. Reexamining the word length effect in visual word recognition: New evidence from the English lexicon project. *Psychonomic Bulletin & Review*, 13(1):45–52, Feb. 2006. 190

1349. A. Newell. *Unified Theories of Cognition*. Harvard University Press, 1991. 18

1350. A. Newell and P. S. Rosenbloom. Mechanisms of skill acquisition and the power law of practice. Technical report, Carnegie Mellon University, Aug. 1982. 33

1351. S. E. Newstead and K. R. Coventry. The role of context and functionality in the interpretation of quantifiers. *European Journal of Cognitive Psychology*, 12(2):243–259, June 2000. 48

1352. G. Nezlek and G. DeHondt. An empirical investigation of gender wage differences in information systems occupations: 1991-2008. In *Proceedings of the 43rd Hawaii International Conference on System Sciences–2010*, HICSS, pages 4059–4068, Jan. 2010. 67

1353. T. H. D. Nguyen, B. Adams, and A. E. Hassan. A case study of bias in bug-fix datasets. In *17th Working Conference on Reverse Engineering*, WCRE'10, pages 259–268, Oct. 2010. 148

1354. V. H. Nguyen and F. Massacci. The (un)reliability of NVD vulnerable versions data: an empirical experiment on Google Chrome vulnerabilities. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, ASIA CCS'13, pages 493–498, May 2013. 148

1355. T. Nicholas. *VC: An American History*. Harvard University Press, June 2019. 89

1356. T. Nichols. A penny saved: Psychological pricing. blog: Gumroad, Oct. 2013. http://blog.gumroad.com/post/64417917582/a-penny-saved-psychological-pricing. 82

1357. W. Nichols. The end to the myth of "Individual Programmer Productivity". *IEEE Software*, 36(5):71–75, Sept.-Oct. 2019. 55

1358. W. R. Nichols, J. D. McHale, D. Sweeney, W. Snavely, and A. Volkman. Composing effective software security assurance workflows. Technical Report CMU/SEI-2018-TR-004, Software Engineering Institute, Carnegie Mellon University, Oct. 2018. 69, 137, 160, 164

1359. A. Nieder. The adaptive value of numerical competence. *Trends in Ecology & Evolution*, 35(7):605–617, July 2020. 18

1360. J. Nielsen and T. K. Landauer. A mathematical model of the finding of usability problems. In *Proceedings of the conference on Human factors in computing systems*, INTERCHI'93, pages 206–213, Apr. 1993. 165, 166

1361. E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. In *Proceedings of the sixth conference on Computer systems*, EuroSys'11, pages 343–356, Apr. 2011. 274

1362. M. J. Nigrini and S. J. Miller. Data diagnostics using second order tests of Benford's law. *Auditing: A Journal of Practice and Theory*, 28(2):305–324, June 2009. 382

1363. D. E. Nikonov and I. A. Young. Overview of beyond-CMOS devices and a uniform methodology for their benchmarking. In *eprint arXiv:cond-mat.mes-hall/1302.0244*, Feb. 2013. 364

1364. J. Ninio and K. A. Stevens. Variations on the Hermann grid: an extinction illusion. *Perception*, 29(10):1209–1217, Oct. 2000. 187

1365. R. E. Nisbett, D. H. Krantz, C. Jepson, and Z. Kunda. The use of statistical heuristics in everyday inductive reasoning. *Psychological Review*, 90(4):339–363, Oct. 1983. 38

1366. NIST??? National vulnerability database. https://nvd.nist.gov, Dec. 2014. 147, 376

1367. S. Nørby. Why forget? On the adaptive value of memory loss. *Perspectives on Psychological Science*, 10(5):551–578, Sept. 2015. 32

1368. P. V. Norden. Resource usage and network planning techniques. In B. V. Dean, editor, *Operations Research in Research and Development*, chapter 5, pages 149–169. John Wiley & Sons, Inc, 1963. 124

1369. W. D. Nordhaus. The progress of computing. Cowles Foundation Discussion Paper No. 1324, Yale University, Sept. 2001. 1

1370. J. A. Norton and F. M. Bass. A diffusion theory model of adoption and substitution for successive generations of high-technology products. *Management Science*, 33(9):1069–1086, Sept. 1987. 83

1371. M. A. Nowak. *Evolutionary Dynamics: Exploring the Equations of Life*. The Belknap press of Harvard University press, 2006. 96

1372. M. A. Nowak and K. Sigmund. Evolution of indirect reciprocity by image scoring. *Nature*, 393(6685):573–577, June 1998. 74

1373. D. Nowroth, I. Polian, and B. Becker. A study of cognitive resilience in a JPEG compressor. In *IEEE International Dependable Systems and Networks With FTCS and DCC*, DSN 2008, pages 32–41, June 2008. 162

1374. H.-C. Nuerk, G. Wood, and K. Willmes. The universal snarc effect: The association between number magnitude and space is amodal. *Experimental Psychology*, 52(3):187–194, 2005. 20, 21

1375. Y. S. Nugroho, H. Hata, and K. Matsumoto. How different are different diff algorithms in Git? Use -histogram for code changes. In *eprint arXiv:cs.SE/1902.02467*, July 2019. 354

1376. R. E. Núñez. No innate number line in the human brain. *Journal of Cross-Cultural Psychology*, 42(4):651–668, 2011. 46

1377. J. M. Nuttin, Jr. Affective consequence of mere ownership: The name letter effect in twelve European languages. *European Journal of Social Psychology*, 17:381–402, 1987. 190

1378. NVIDIA. *CUDA CUBLAS Library*. NVIDIA Corporation, CA, USA, 3.1 edition, Aug. 2010. 357

1379. M. Oaksford and N. Chater. A rational analysis of the selection task as optimal data selection. *Psychological Review*, 101(4):608–631, Oct. 1994. 42

1380. K. Oberauer, S. Lewandowsky, E. Awh, G. D. A. Brown, A. Conway, N. Cowan, C. Donkin, S. Farrell, G. J. Hitch, M. Hurlstone, W. J. Ma, C. C. Morey, D. E. Nee, J. Schweppe, E. Vergauwe, and G. Ward. Benchmarks for models of short term and working memory. *Psychological Bulletin*, 144(9):885–958, Sept. 2018. 28

1381. K. Oberauer, H.-M. Süß, O. Wilhelm, and W. W. Wittmann. The multiple faces of working memory: Storage, processing, supervision, and coordination. *Intelligence*, 31(2):167–193, Mar.-Apr. 2003. 29

1382. M. Ochodek, J. Nawrocki, and K. Kwarciak. Simplifying effort estimation based on use case points. *Information and Software Technology*, 53(3):200–213, Mar. 2011. 125

1383. O. O. Odeh, A. M. Featherstone, and J. S. Bergtold. Reliability of statistical software. *American Journal of Agricultural Economics*, 92(5):1472–1489, Sept. 2010. 14

1384. OECD. *OECD Digital Economy Outlook 2015*. OECD Publishing, 2015. 58

1385. Office for National Statistics, UK. GFCF estimates for computer software purchases, own account computer software. website, June 2017. http://www.ons.gov.uk/ons/rel/bus-invest/business-investment/index.html. 4, 101

1386. C. Ogden. Killed by Google. https://killedbygoogle.com, Nov. 2018. 61, 95, 96

1387. S. Ogilvie. *The European Guilds: An Economic Analysis*. Princeton University Press, Feb. 2019. 75, 95

1388. J. Oh, D. Batory, M. Heule, M. Myers, and P. Gazzillo. Uniform sampling from Kconfig feature models. Technical Report 19-02, The University of Texas at Austin, Department of Computer Science, 2019. 251

1389. S. Ohlsson. The learning curve for writing books: Evidence from Professor Asimov. *Psychological Science*, 3(6):380–382, Nov. 1992. 37

1390. M. Ohm, H. Plate, A. Sykosch, and M. Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *eprint arXiv:cs.CR/2005.09535*, May 2020. 157

1391. H. Ohtsuki, C. Hauert, E. Lieberman, and M. A. Nowak. A simple rule for the evolution of cooperation on graphs and social networks. *Nature*, 441(7092):502–505, May 2006. 74

1392. H. Ohtsuki and Y. Iwasa. How should we define goodness?–reputation dynamics in indirect reciprocity. *Journal of Theoretical Biology*, 231(1):107–120, Nov. 2004. 74

1393. P. Oladimeji. Devices, errors and improving interaction design-A case study using an infusion pump. Thesis (m.res.), Department of Computer Science, Swansea University, Oct. 2008. 243

1394. A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN'07, pages 575–584, June 2007. 163

1395. P. Oliver. Experiences in building and using compiler validation systems. In R. Merwin and J. Zanca, editors, *AFIPS Conference Proceedings*, Volume 48, pages 1051–1057, June 1979. 167

1396. S. O'Mahony. *Can Medicine be Cured? The Corruption of a Profession*. Head of Zeus Ltd, Feb. 2019. 10

1397. O*NET OnLine. website, July 2019. https://www.onetonline.org. 104

1398. T. Open Group. The Austin common standards revision group. http://austingroupbugs.net, July 2017. 160

1399. Open Science Collaboration. Estimating the reproducibility of psychological science. *Science*, 349(6251):aac4716, Aug. 2015. 11

1400. OpenCorporates. UK registered company data. https://opencorporates.com, Mar. 2015. 103

1401. OpenRefine. website, Oct. 2014. http://openrefine.org. 374

1402. OpenSignal. Android fragmentation visualized (august 2015). Technical Report ???, OpenSignal, Aug. 2015. 221, 222, 375

1403. A. Orlitsky, A. T. Suresh, and Y. Wu. Optimal prediction of the number of unseen species. *PNAS*, 113(47):13283–13288, Nov. 2016. 100

1404. N. Osaka. Eye fixation and saccade during kana and kanji text reading: Comparison of English and Japanese text processing. *Bulletin of the Psychonomic Society*, 27(6):548–550, 1989. 26

1405. A. T. Oskarsson, L. V. Boven, G. H. McClelland, and R. Hastie. What's next? Judging sequences of binary events. *Psychological Bulletin*, 135(2):262–285, 2009. 19, 49

1406. H. Osman, M. Leuenberger, M. Lungu, and O. Nierstrasz. Tracking null checks in open-source Java systems. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, SANER'16, pages 304–313, Mar. 2016. 199

1407. E. Ostrom, J. Walker, and R. Gardner. Covenants with and without a sword: Self-Governance is possible. *The American Political Science Review*, 86(2):404–417, June 1992. 75

1408. L. M. Ottenstein, V. B. Schneider, and M. H. Halstead. Predicting the number of bugs expected in a program module. Technical Report CSD-TR 205, Purdue University, Oct. 1976. 194

1409. M. A. Oumaziz, A. Charpentier, J.-R. Falleri, and X. Blanc. Documentation reuse: Hot or not? An empirical study. In *16th International Conference on Software Reuse*, ICSR 2017, pages 12–27, May 2017. 159

1410. G. L. Ourada. Software cost estimating models: A calibration, validation, and comparison. Thesis (m.s.), Air Force Institute of Technology, USA, Dec. 1991. 6, 124

1411. C. Overney, J. Meinicke, C. Kästner, and B. Vasilescu. How to not get rich: An empirical study of donations in Op€n $our¢e. In *42nd International Conference on Software Maintenance*, ICSE'20, page ???, July 2020. 89

1412. S. Owsowitz and A. Sweetland. Factors affecting coding errors. Research Memorandum RM-4346-PR, The RAND Corporation, Apr. 1965. 21

1413. S. C. Özbek. *Introducing Innovations into Open Source Projects*. PhD thesis, Freie Universität Berlin, Aug. 2010. 128

1414. A. Ozment and S. E. Schechter. Milk or wine: Does software security improve with age? In *USENIX Security Symposium*, SEC'06, pages 93–104, July-Aug. 2006. 140

1415. P. Padfield. *Battleship*. Thistle Publishing, 2015. 3

1416. R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *Proceedings of the 3rd USENIX conference on Offensive technologies*, WOOT'09, pages 2–2, Aug. 2009. 144

1417. N. Palix, J. Lawall, and G. Muller. Tracking code patterns over multiple software versions with Herodotus. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD'10, pages 169–180, Mar. 2010. 150

1418. N. Palix, S. Saha, G. Thomas, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. Technical Report RR-7357, Institut National de Recherche en Informatique et en Automatique, Aug. 2010. 143

1419. J. Pallister, S. Hollis, and J. Bennett. Identifying compiler options to minimise energy consumption for embedded platforms. In *eprint arXiv:cs.PF/1303.6485*, Aug. 2013. 361

1420. E. M. Palmer, T. S. Horowitz, A. Torralba, and J. M. Wolfe. What are the shapes of response time distributions in visual search? *Journal of Experimental Psychology: Human Perception and Performance*, 37(1):58–71, Feb. 2011. 25, 26

1421. S. E. Palmer. *Vision Science: Photons to Phenomenology*. The MIT Press, 1999. 25

1422. H.-Y. Pan, A. Chao, and W. Foissner. A nonparametric lower bound for the number of species shared by multiple communities. *Journal of Agricultural, Biological, and Environmental Statistics*, 14(4):452–468, Dec. 2009. 100

1423. K. Pan. *Using Evolution Patterns to Find Duplicated Bugs*. PhD thesis, Department of Computer Science, University of California at Santa Cruz, Oct. 2006. 158

1424. T. Pani. Loop patterns in C programs. Thesis (m.s.), Fakultät für Informatik der Technischen Universität Wien, Dec. 2013. 178, 200

1425. R. Parker and B. Grimm. Recognition of business and government expenditures for software as investment: Methodology and quantitative impacts, 1959-98. In *BEA Advisory Committee meeting*, May 2000. 57

1426. A. Parkhomenko, A. Redkina, and O. Maslivets. Estimating hedonic price indexes for personal computers in Russia. MPRA Paper No. 5019, Higher School of Economics, Jan. 2007. 82

1427. C. Parnin, C. Bird, and E. Murphy-Hill. Adoption and use of Java generics. *Empirical Software Engineering*, 18(6):1047–1089, Dec. 2013. 181

1428. F. N. Parr. An alternative to the Rayleigh curve model for software development effort. *IEEE Transactions on Software Engineering*, SE-6(3):291–296, May 1980. 124

1429. H. E. Pashler. *The Psychology of Attention*. The MIT Press, 1999. 24

1430. L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wąsowski, and P. Borba. Coevolution of variability models and related artefacts: A case study of the Linux kernel. In *Proceedings of the 17th International Software Product Line Conference*, SPLC'13, pages 91–100, Apr. 2013. 93

1431. A. Patel. Auditors' belief revision: Recency effects of contrary and supporting audit evidence and source reliability. Technical Report 2001-1, Department of AFM/SSE, University of South Pacific, June 2001. 36

1432. M. R. Patterson. *Antitrust Law in the New Economy :Google, Yelp, LIBOR, and the Control of Information*. Harvard University Press, 2017. 90, 98

1433. F. M. Paulus, L. Rademacher, T. A. J. Schäfer, L. Müller-Pinzler, and S. Krach. Journal impact factor shapes scientists' reward signal in the prospect of publication. *PLoS ONE*, 10(11):e0142537, Nov. 2015. 9

1434. A. Pavese and C. Umiltà. Symbolic distance between numerosity and identity moulates Stroop-like interference. *Journal of Experimental Psychology: Human Perception and Performance*, 24(5):1535–1545, 1998. 28

1435. E. Pavese, E. Soremekun, N. Havrikov, L. Grunske, and A. Zeller. Inputs from hell: Generating uncommon inputs from common samples. In *eprint arXiv:cs.SE/1812.07525*, Dec. 2018. 169

1436. J. W. Payne, J. R. Bettman, and E. J. Bettman. *The Adaptive Decision Maker*. Cambridge University Press, 1993. 51

1437. G. Paz-y-Miño C, A. B. Bond, A. C. Kamil, and R. P. Balda. Pinyon jays use transitive inference to predict social dominance. *Nature*, 430:778–781, Aug. 2004. 44

1438. R. D. Pea. Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, 2(1):25–36, Feb. 1986. 174

1439. J. Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2000. 45

1440. R. K. Pearson. The problem of disguised missing data. *ACM SIGKDD Explorations Newsletter*, 8(1):83–92, June 2006. 377

1441. Y. Peers. *Econometric Advances in Diffusion Models*. PhD thesis, Erasmus University, Rotterdam, Dec. 2011. 82, 83

1442. E. Pek. *Corpus-based Empirical Research in Software Engineering*. PhD thesis, Department of Computer Science, Universität Koblenz-Landau, Oct. 2013. 363

1443. D. G. Pelli, C. W. Burns, B. Farell, and D. C. Moore-Page. Feature detection and letter identification. *Vision Research*, 46(28):4646–4674, 2006. 27

1444. J. Peltokorpi and E. Niemi. Effects of group size and learning on manual assembly performance: an experimental study. *International Journal of Production Research*, 57(2):452–469, 2019. 138

1445. E. Peltonen, E. Lagerspetz, P. Nurmi, and S. Tarkoma. Energy modeling of system settings: A crowdsourced approach. In *IEEE International Conference on Pervasive Computing and Communications*, PerCom'15, pages 37–45, Mar. 2015. 364

1446. N. Pennington. Comprehension strategies in programming. In G. Olson, S. Shepard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, chapter 7, pages 100–113. Ablex Publishing Corporation, 1987. 182

1447. N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3):295–341, July 1987. 32

1448. B. T. Pentland and H. H. Rueter. Organizational routines as grammars of action. *Administrative Science Quarterly*, 39(3):484–510, Sept. 1994. 102

1449. C. Perez. *Technological Revolutions and Financial Capital: The Dynamics of Bubbles and Golden Ages*. Edward Elgar Publishing, 2003. 3, 5

1450. D. Perez and B. Livshits. Smart contract vulnerabilities: Does anyone care? In *eprint arXiv:cs.CR/1902.06710*, Feb. 2019. 147

1451. D. E. Perry and W. M. Evangelist. An empirical study of software interface faults – An update. In *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences, Vol II*, HICSS, pages 113–126, Jan. 1987. 7, 147

1452. D. E. Perry, N. A. Staudenmayer, and L. G. Votta, Jr. Understanding and improving time usage in software development. In A. Fuggetta and A. L. Wolf, editors, *Trends in Software Process*, chapter 5, pages 111–135. John Wiley & Sons, Mar. 1995. 354

1453. D. E. Perry and C. S. Stieg. Software faults in evolving a large, real-time system: a case study. In *Proceedings of the 1993 European Software Engineering Conference*, pages 48–67, Sept. 1993. 147

1454. R. Perugupalli. Empirical assessment of architecture-based reliability of open-source software. Thesis (m.s.), Department of Computer Science and Electrical Engineering, West Virginia University, May 2004. 244

1455. H. Petroski. *Design Paradigms: Case Histories of Error and Judgment in Engineering*. Cambridge University Press, 1994. 144

1456. C. Peukert. Switching costs and information technology: The case of IT outsourcing. In 1^{st} *ICT Conference Munich on ICT and Economic Growth*, Nov. 2010. 136

1457. A. Pewsey, M. Neuhäuser, and G. D. Ruxton. *Circular Statistics in R*. Oxford University Press, 2013. 340, 341, 342

1458. P. M. Pexman and M. J. Yap. Individual differences in semantic processing: Insights from the Calgary semantic decision project. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 44(7):1091–1112, Feb. 2018. 189

1459. R.-H. Pfeiffer. What constitutes software? An empirical, descriptive study of artifacts. In *msr*, MSR 2020, page ???, June 2020. 174

1460. S. A. Phatak, A. Lovitt, and J. B. Allen. Consonant confusions in white noise. *Journal of the Acoustic Society of America*, 124(2):1220–1233, Aug. 2008. 191

1461. A. Phillips. *Technology and Market Structure: A Study of the Aircraft Industry*. Heath Lexington Books, 1972. 95

1462. C. Phillips. *Order and Structure*. PhD thesis, M.I.T., Aug. 1996. 30

1463. M. Phister, Jr. *Data Processing Technology and Economics*. Santa Monica Publishing Company and Digital Press, second edition, 1979. 6, 101

1464. S. T. Piantadosi. Zipf's word frequency law in natural language: a critical review and future directions. *Psychonomic Bulletin & Review*, 21(5):1112–1130, Oct. 2014. 202

1465. S. T. Piantadosi. A rational analysis of the approximate number system. *Psychonomic Bulletin & Review*, 23(3):877–886, June 2016. 46

1466. R. Pieters and L. Warlop. Visual attention during brand choice: The impact of time pressure and task motivation. *International Journal of Research in Marketing*, 16:1–16, 1999. 26

1467. D. J. Pigott and B. M. Axtens. Online historical encyclopedia of programming languages. `http://hopl.info`, 2015. 109

1468. R. S. Pindyck. Investments of uncertain cost. *Journal of Financial Economics*, 34(1):53–76, Aug. 1993. 62

1469. J. Pipitone. Software quality in climate modelling. Thesis (m.s.), Department of Computer Science, University of Toronto, 2010. 152

1470. A. M. Pires and C. Amado. Interval estimators for a binomial proportion: Comparison of twenty methods. *REVSTAT-Statistical Journal*, 6(2):165–197, June 2008. 264

1471. P. Pirolli. *Information Foraging Theory: Adaptive Interaction with Information*. Oxford University Press, May 2007. 178

1472. D. J. Pittenger. Measuring the MBTI . . . And coming up short. *Journal of Career Planning and Employment*, 54(1):48–52, Nov. 1993. 50

1473. PK. How many developers are there in America, and where do they live? website, Apr. 2019. `https://dqydj.com/number-of-developers-in-america-and-per-state`. 99

1474. J. Plamondon. Effective evangelism: Joe comes, riley paint, inc., skeffingon's formal wear, inc., patricia anne larsen vs. microsoft corporation. Plaintiff's Exhibit 3096, IOWA District Court for Polk County, Jan. 2000. 81

1475. A. Pluchino, A. Rapisarda, and C. Garofalo. The Peter principle revisited: A computational study. In *eprint arXiv:physics.soc-ph/0907.0455v3*, Oct. 2009. 67

1476. T. Plum. *Reliable data structures in C*. Plum Hall, 1985. 179

1477. T. Plum. *C Programming guidelines*. Plum Hall, 1989. 179

1478. I. P. L. Png. On the reliability of software piracy statistics. *Electronic Commerce Research and Applications*, 9(5):365–373, Sept.-Oct. 2010. 84

1479. A. Pogačnik and A. Črnič. iReligion: Religious elements of the Apple phenomenon. *The Journal of Religion and Popular Culture*, 26(3):353–364, Sept.-Nov. 2014. 70

1480. C. Poivey, J. L. Barth, K. A. LaBel, G. Gee, and H. Safren. In-flight observations of long-term single-event effect (SEE) performance on Orbview-2 solid state recorders (SSR). In *2003 IEEE Radiation Effects Data Workshop*, pages 102–107, July 2003. 221

1481. C. Politowski, F. Petrillo, G. C. Ullmann, J. de Andrade Werly, and Y.-G. Guéhéneu. Dataset of video game development problems. In *eprint arXiv:cs.SE/2001.00491*, Jan. 2020. 121

1482. R. Pollack. How to believe a machine-checked proof. In G. Sambin and J. M. Smith, editors, *Twenty Five Years of Constructive Type Theory*, chapter 11, pages 205–220. Oxford University Press, Oct. 1998. 145

1483. A. Pollatsek, E. D. Reichle, and K. Rayner. Tests of the E-Z reader model: Exploring the interface between cognition and eye-movement control. *Cognitive Psychology*, 52(1):1–56, Feb. 2006. 26

1484. G. Poo-Caamaño and D. M. German. Software patents: A replication study. In *Proceedings of the 11th International Symposium on Open Collaboration*, OpenSym'15, pages 5:1–5:4, Aug. 2015. 64

1485. D. Pope and U. Simonsohn. Round numbers as goals: Evidence from baseball, SAT takers, and the lab. *Psychological Science*, 22(1):71–79, Jan. 2011. 47

1486. K. R. Popper. *Conjectures and Refutations*. Routledge, 1969. 23

1487. A. Porter, H. Siy, A. Mockus, and L. Votta. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering Methodology*, 7(1):41–79, Jan. 1998. 165, 299, 356

1488. M. E. Porter. *Competitive Advantage: Creating and Sustaining Superior Performance*. First Free Press, 1985. 81

1489. M. E. Porter. The five competitive forces that shape strategy. *Harvard Business Review*, 86(1):78–93, Jan. 2008. 58, 96

1490. R. D. Portugal and B. F. Svaiter. Weber-Fechner law and the optimality of the logarithmic scale. *Minds & Machines*, 21(1):73–81, Feb. 2011. 56

1491. A. S. Posamentier and I. Lehmann. *Magnificent mistakes in mathematics*. Prometheus books, 2013. 144

1492. D. E. Post and R. P. Kendall. Software project management and quality engineering practices for complex, coupled multi-physics, massively parallel computational simulations: Lessons learned from ASCI. Report LA-UR-03-1274 Rev. 2, Los Alamos National Laboratory, Mar. 2004. 107

1493. A. Potanin, M. Damitio, and J. Noble. Are your incoming aliases really necessary? Counting the cost of object ownership. In *Proceedings of the 2013 International Conference on Software*, ICSE'13, pages 742–751, May 2013. 267

1494. E. M. Pothos and N. Chater. Rational categories. In *Proceedings of the Twentieth Annual Conference of the Cognitive Science Society*, pages 848–853, 1998. 38

1495. M. C. Potter, A. Moryadas, I. Abrams, and A. Noel. Word perception and misperception in context. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 19(1):3–22, 1993. 191

1496. J. Potts, J. Hartley, L. Montgomery, C. Neylon, and E. Rennie. A journal is a club: A new economic model for scholarly publishing. Working Paper n. 2763975, Australian universities, Apr. 2016. 9

1497. A. L. Powell. *Right on Time: Measuring, Modelling and Managing Time-Constrained Software Development*. PhD thesis, Department of Computer Science, University of York, Aug. 2001. 117, 330, 331

1498. D. A. Powner and K. A. Rhodes. Business systems modernization: IRS needs to complete recent efforts to develop policies and procedures to guide requirements development and management. Technical Report GAO-06-310, United States Government Accountability Office, Mar. 2006. 131

1499. M. Pradel. *Program Analyses for Automatic and Precise Error Detection*. PhD thesis, ETH Zurich, 2012. 153, 155

1500. M. Pradel and T. Sen. DeepBugs: A learning approach to name-based bug detection. In *Proceedings of the ACM on Programming Languages*, OOPSLA'18, page 147, Nov. 2018. 191

1501. V. Prasad, A. Vandross, C. Toomey, M. Cheung, J. Rho, S. Quinn, S. J. Chacko, D. Borkar, V. Gall, S. Selvaraj, N. Ho, and A. Cifu. A decade of reversal: An analysis of 146 contradicted medical practices. *Mayo Clinical Proceedings*, 88(8):790–798, Aug. 2013. 10

1502. L. Prechelt. The 28:1 Grant/Sackman legend is misleading, or: How large is interpersonal variation really? Technical Report iratr-1999-18, Universität Karlsruhe, 1999. 8, 55, 218

1503. L. Prechelt. Plat_Forms 2007: The web development platform comparison – evaluation and results. Technical Report B-07-10, Institut für Informatik, Freie Universität Berlin, June 2007. 126, 127, 131, 133, 210

1504. L. Prechelt, D. Graziotin, and D. M. Fernández. On the status and future of peer review in software engineering. In *eprint arXiv:cs.SE/1706.07196*, June 2017. 9

1505. L. Prechelt and W. F. Tichy. A controlled experiment measuring the effect of procedure argument type checking on programmer productivity. *IEEE Transactions on Software Engineering*, 24(4):302–312, Apr. 1998. 192

1506. L. Prechelt, F. Zieris, and H. Schmeisky. Difficulty factors of obtaining access for empirical studies in industry. In *Proceedings of the Third International Workshop on Conducting Empirical Studies in Industry*, CESI'15, pages 19–25, May 2015. 6

1507. L. S. Premo and S. L. Kuhn. Modeling effects of local population extinctions on cultural change and diversity in the paleolithic. *PLoS ONE*, 5(12):e15582, Dec. 2010. 72, 223

1508. R. A. Prentice and J. H. Langmore. Beware of vaporware: Product hype and the securities fraud liability of high-tech companies. *Harvard Journal of Law & Technology*, 8(1):1–74, Oct.-Dec. 1994. 73

1509. C. C. Presson and D. R. Montello. Updating after rotational and translational body movements: coordinate structure of perspective space. *Perception*, 23:1447–1455, 1994. 20

1510. T. Preston-Werner. Semantic versioning 2.0.0. website, July 2019. `https://semver.org`. 113

1511. D. Pritchard. Frequency distribution of error messages. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU 2015, pages 1–8, Oct. 2015. 160

1512. V. Propp. *Morphology of the Folktale*. University of Texas Press, second edition, 1968. English translation by Laurence Scott. 180

1513. J. Prümper, D. Zapf, F. C. Brodbeck, and M. Frese. Some surprising differences between novice and expert errors in computerized office work. *Behaviour & Information Technology*, 11(6):319–328, 1992. 143

1514. D. Pukhkaiev. Energy-efficient benchmarking for energy-efficient software. Thesis (m.s.), Technische Universität, Dresden, Dec. 2015. 359

1515. R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, June 2005. 160, 161

1516. L. H. Putnam. A general empirical solution to the macro software sizing and estimating problem. *IEEE Transactions on Software Engineering*, SE-4(4):345–361, July 1978. 124

1517. L. H. Putnam and W. Myers. *Measures for Excellence: Reliable software on time, within budget*. Prentice-Hall, Inc, 1992. 226

1518. PwC. Converging forces are building that could re-shape the entire industry. Global 100 software leaders, PwC Technology Institute, May 2013. 84

1519. PwC. The growing importance of apps and services. Global 100 software leaders, PwC Technology Institute, Mar. 2014. 84

1520. PwC. Digital intelligence conquers the world below and the cloud above. Global 100 software leaders, PwC Technology Institute, 2016. 84

1521. PwC. IPO review full-year and Q4 2015. Global technology, PwC Technology Institute, Feb. 2016. 89

1522. Z. Pylyshyn. Is vision continuous with cognition? The case for cognitive impenetrability of visual perception. *Behavioral and Brain Sciences*, 22(3):341–423, 1999. 25

1523. X. Qu. *Configuration aware prioritization for regression testing*. PhD thesis, The Graduate College at the University of Nebraska, Apr. 2010. 170

1524. S. Qualline. *C Elements of Style*. M&T Books, 1992. 179

1525. R. Queiroz, L. Passos, M. T. Valente, C. Hunsen, S. Apel, and K. Czarnecki. The shape of feature code: an analysis of twenty C-preprocessor-based systems. *Journal on Software and Systems Modeling*, 16(1):77–96, Feb. 2017. 315, 316

1526. R Core Team. R language definition. Technical Report 3.3.1, R Foundation for Statistical Computing, June 2016. 383

1527. R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2019. ISBN 3-900051-07-0. 383, 384

1528. H. Rabinowitz and C. Schaap. *Portable C*. Prentice-Hall, Inc, 1990. 179

1529. J. J. Rachlinski, A. J. Wistrich, and C. Guthrie. Can judges make reliable numeric judgments? Distorted damages and skewed sentences. *Indiana Law Journal*, 90(2):695–739, Apr. 2015. 47

1530. J. W. Radatz. Analysis of IV & V data. Technical Report RADC-TR-81-145, Rome Air Development Center, Griffiss Air Force Base, June 1981. 157

1531. G. Radden and R. Dirven. *Cognitive English Grammar*. John Benjamins Publishing Company, 2007. 173

1532. D. Raffo, J. Settle, and W. Harrison. Investigating financial measures for planning software IV&V. Technical Report TR-99-05, Portland State University, 1999. 60

1533. C. Ragkhitwetsagul, J. Krinke, and D. Clark. A comparison of code similarity analysers. *Empirical Software Engineering*, 23(4):2464–2519, Aug. 2018. 196

1534. F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? In *Proceedings of the 7th International Workshop on Mining Software Repositories*, MSR'10, pages 72–81, May 2010. 7

1535. M. T. Rahman, E. Shihab, and P. C. Rigby. The modular and feature toggle architectures of Google Chrome. *Empirical Software Engineering*, 24(2):826–853, July 2019. 193, 194

1536. J. Ranade and A. Nash. *The Elements of C Programming Style*. McGraw-Hill, Inc, 1992. 179

1537. A. Rashid, H. Chivers, G. Danezis, E. Lupu, and A. Martin. CyBOK: The cyber security body of knowledge. Technical Report 1.0, The National Cyber Security Centre, UK, Oct. 2019. 149

1538. R. Ratcliff, G. McKoon, and P. Gomez. A diffusion model account of the lexical decision task. *Psychological Review*, 111(1):159–182, Jan. 2004. 20

1539. R. Ratcliff, P. L. Smith, S. D. Brown, and G. McKoon. Diffusion decision model: Current issues and history. *Trends in Cognitive Sciences*, 20(4):260–281, Apr. 2016. 20

1540. B. Ray. *Analysis of Cross-System Porting and Porting Errors in Software Projects*. PhD thesis, University of Texas at Austin, Aug. 2013. 93, 94

1541. B. Ray, M. Wilcox, and C. Voskoglou. Developer economics | State of the developer nation q3 2015. State of the Nation 9th edition, VisionMobile, July-Sept. 2015. 110

1542. K. Rayner. Eye movements and attention in reading, scene perception, and visual search. *The Quarterly Journal of Experimental Psychology*, 62(8):1457–1506, 2009. 26

1543. K. Rayner. Eye movements in reading: Models and data. *Journal of Eye Movement Research*, 2(5):1–10, Apr. 2009. 25

1544. N. M. Razali and Y. B. Wah. Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests. *Journal of Statistical Modeling and Analytics*, 2(1):21–33, 2011. 259

1545. J. Reason. *Human Error*. Cambridge University Press, 1990. 143, 157

1546. A. S. Reber and S. M. Kassin. On the relationship between implicit and explicit modes in the learning of a complex rule structure. *Journal of Experimental Psychology: Human Learning and Memory*, 6(5):492–502, 1980. 34

1547. J. L. Recón. Building skyscrapers, and spending on major projects. https://github.com/jlricon/open_nintil, Oct. 2018. https://nintil.com/2018/10/07/building-skyscrapers-and-spending-on-major-projects. 122

1548. B. Regnell, M. Höst, J. N. och Dag, P. Beremark, and T. Hjelm. An industrial case study on distributed prioritisation in market-driven requirements engineering for packaged software. *Requirements Engineering*, 6(1):51–62, Apr. 2001. 52, 132

1549. E. D. Reichle, T. Warren, and K. McConnell. Using E-Z reader to model the effects of higher-level language processing on eye movements during reading. *Psychonomic Bulletin & Review*, 16(1):1–21, Feb. 2009. 26

1550. R. J. Reid. The Reid list of the first course language for computer science majors. http://www.csee.wvu.edu/~vanscoy/reid.htm, Aug. 2002. 111

1551. J. Reimer. Computer smartphone and tablet marketshare: 1975-2012. website, Dec. 2012. http://jeremyreimer.com/m-item.lsp?i=137. 3, 89

1552. G. A. Reis III. *Software Modulated Fault Tolerance*. PhD thesis, Department of Electrical Engineering, Princeton University, June 2008. 162

1553. G. Remillard. Implicit learning of second-, third-, and fourth-order adjacent and nonadjacent sequential dependencies. *The Quarterly Journal of Experimental Psychology*, 61(3):400–424, Apr. 2008. 27

1554. R. W. Remington, H. W. H. Yuen, and H. Pashler. With practice, keyboard shortcuts become faster than menu selection: A crossover interaction. *Journal of Experimental Psychology: Applied*, 22(1):95–106, 2016. 38

1555. Research Councils UK. RCUK policy on open access and supporting guidance. Technical report, RCUK, Mar. 2013. 9

1556. A. Rice, E. Aftandilian, C. Jaspan, E. Johnston, M. Pradel, and Y. Arroyo-Paredes. Detecting argument selection defects. *Proceedings of the ACM on Programming Languages*, 1(1):104, Oct. 2017. 190

1557. G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do A large-scale study of the use of eval in JavaScript applications. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 52–78, July 2011. 195

1558. G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'10, pages 1–12, June 2010. 195

1559. R. Richardson. 2008 CSI computer crime & security survey. Technical report, Computer Security Institute, Aug. 2008. 150

1560. D. F. Rico. Short history of software methods. http://davidfrico.com/rico04e.pdf, July 2004. 127

1561. R. K. Ridgway. Compiling routines. In *Proceedings of the 1952 ACM national meeting (Toronto)*, ACM'52, pages 1–5, Sept. 1952. 109

1562. R. Riesen, K. Ferreira, J. Stearley, R. Oldfield, J. H. Laros III, K. Pedretti, and R. Brightwell. Redundant computing for exascale systems. Technical Report SAND2010-8709, Sandia National Laboratories, Dec. 2010. 162

1563. M. Rigger, S. Marr, B. Adams, and H. Mössenböck. Understanding GCC builtins to develop better tools. In *eprint arXiv:cs.PL/1907.00863*, July 2019. 111

1564. M. Rigger, S. Marr, S. Kell, D. Leopoldseder, and H. Mössenböck. An analysis of x86-64 inline assembly in C programs. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'18, pages 84–99, Mar. 2018. 198

1565. M. Rinard, C. Cadar, and H. H. Nguyen. Exploring the acceptability envelope. In *Companion to the 20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'05, pages 21–30, Oct. 2005. 143

1566. M. Ringelmann. Recherches sur les moteurs animés: Travail de l'homme. *Annales de l'Institut National Agronomique*, 2(XII):1–40, 1913. 76

1567. J. S. Riordon. An evolution dynamics model of software systems development. In B. Elkins and L. Hunt, editors, *Software Phenomenology Working Papers of the Software Life Cycle Management Workshop*, pages 339–360. US Army Institute for Research in Management Information and Computer SCience, Aug. 1997. 134

1568. R. Robbes, D. Róthlisberger, and É. Tanter. Object-oriented software extensions in practice. *Empirical Software Engineering*, 20(3):745–782, June 2015. 204, 205

1569. M. J. Roberts, D. J. Gilmore, and D. J. Wood. Individual differences and strategy selection in reasoning. *British Journal of Psychology*, 88:473–492, 1997. 42

1570. S. Roberts and J. Winters. Linguistic diversity and traffic accidents: Lessons from statistical studies of cultural traits. *PLoS ONE*, 8(8):e70902, Aug. 2013. 263

1571. D. E. Robinson. Fashions in shaving and trimming of the beard: The men of the Illustrated London News, 1842-1972. *American Journal of Sociology*, 81(5):1133–1141, Mar. 1976. 8

1572. G. Robles and J. M. González-Barahona. A comprehensive study of software forks: Dates, reasons and outcomes. In *The 8th International Conference on Open Source Systems*, OSS 2012, pages 1–14, Sept. 2012. 92

1573. G. Robles, I. Herraiz, D. M. Germán, and D. Izquierdo-Cortázar. Modification and developer metrics at the function level: Metrics for the study of the evolution of a software project. In *3rd International Workshop on Emerging Trends in Software Metrics*, WETSoM'12, pages 49–55, June 2012. 206, 207, 208

1574. G. Robles, L. A. Reina, A. Serebrenik, B. Vasilescu, and J. M. González-Barahona. FLOSS 2013: A survey dataset about free software contributors: Challenges for curating, sharing, and combining. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR'14, pages 396–399, May 2014. 214, 371

1575. E. Rodrigues, Jr. and R. Terra. How do developers use dynamic features? The case of Ruby. *Computer Languages, Systems & Structures*, 53:73–89, Sept. 2018. 199

1576. W. H. Roetzheim. When the software becomes a nightmare: Dealing with failed projects. *Business Law Today*, 13(6):42–48, July-Aug. 2004. 130

1577. R. D. Rogers and S. Monsell. Costs of a predictable switch between simple cognitive tasks. *Journal of Experimental Psychology: General*, 124(2):207–231, 1995. 24

1578. M. Rönkkö, O.-P. Mutanen, N. Koivisto, J. Ylitalo, J. Peltonen, A.-M. Touru, S. Hyrynsalmi, P. Poikonen, O. Junna, J. Ali-Yrkkö, A. Valtakoski, Y. Huang, and J. Kantola. The finnish software industry in 2007. National Software Industry Survey 2008, Software Business Lab, 2008. 58

1579. E. Rosch, C. B. Mervis, W. D. Gray, D. M. Johnson, and P. Boyes-Braem. Basic objects in natural categories. *Cognitive Psychology*, 8(3):382–439, July 1976. 39

1580. L. Rosen. *Open Source Licensing: Software Freedom and Intellectual Property Law*. Prentice Hall PTR, July 2004. 64

1581. M. Rosenfelder. *The Language Construction Kit*. Yonagu Books, 2010. 109

1582. A. Ross. *No-Collar: The Humane Workplace and its Hidden Costs*. Temple University Press, 2003. 103

1583. B. H. Ross and G. L. Murphy. Food for thought: Cross-classification and category organization in a complex real-world domain. *Cognitive Psychology*, 38(4):495–552, June 1999. 350

1584. L. Ross, M. R. Lepper, and M. Hubbard. Perseverance in self-perception and social perception: Biased attributional processes in the debeliefing paradigm. *Journal of Personality and Social Psychologs*, 32(5):880–892, 1975. 36

1585. B. Rossi, B. Russo, and G. Succi. Path dependent stochastic models to detect planned and actual technology use: A case study of openoffice. *Information & Software Technology*, 53(11):1209–1226, Nov. 2011. 98

1586. J. Rost and R. L. Glass. *The Dark Side of Software Engineering: Evil on Computing Projects*. John Wiley & Sons, Inc, 2011. 118, 131

1587. V. Rothberg, N. Dintzner, A. Ziegler, and D. Lohmann. Feature models in Linux-from symbols to semantics. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS'16, pages 65–72, Jan. 2016. 93

1588. B. F. Roukema. A first-digit anomaly in the 2009 Iranian presidential election. In *eprint arXiv:stat.AP/0906.2789*, June 2013. 382

1589. G. Rousseau, R. Di Cosmo, and S. Zacchiroli. Software provenance tracking at the scale of public source code. HAL Id: hal-02543794, HAL archives-ouvertes.fr, Apr. 2020. 7

1590. E. G. Roy, D. C. Quintero, J. R. Hurley, A. Cierny, and D. Norcia. Plaintiffs, vs. SAMSUNG TELECOMMUNICATIONS AMERICA, LLC, a New York Corporation, and SAMSUNG ELECTRONICS AMERICA, INC., a New Jersey Corporation, Defendants. PLAINTIFF'S NOTICE OF UNOPPOSED MOTION AND UNOPPOSED MOTION FOR PRELIMINARY APPROVAL OF CLASS ACTION SETTLEMENT; MEMORANDUM OF POINTS AND AUTHORITIES CASE NO. 3:14-cv-582-JD, UNITED STATES DISTRICT COURT NORTHERN DISTRICT OF CALIFORNIA, Oct. 2019. 362

1591. M. M. Roy, N. J. S. Christenfeld, and C. R. M. McKenzie. Underestimating the duration of future events: Memory incorrectly used or memory bias? *Psychological Bulletin*, 131(5):738–756, 2005. 53

1592. W. W. Royce. Managing the development of large software systems. In *Technical Papers of Western Electronic Show and Convention*, WesCon, pages 1–9, Aug. 1970. 127

1593. P. Royston, D. G. Altman, and W. Sauerbrei. Dichotomizing continuous predictors in multiple regression: a bad idea. *Statistics in Medicine*, 25(1):127–141, Jan. 2006. 300

1594. D. C. Rubin, S. Hinton, and A. Wenzel. The precise time course of retention. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 25(5):1161–1176, Sept. 1999. 33, 34

1595. D. C. Rubin and A. E. Wenzel. One hundred years of forgetting: A quantitative description of retention. *Psychological Review*, 103(4):734–760, Oct. 1996. 33

1596. C. Rubio-González and B. Libit. Expect the unexpected: Error code mismatches between documentation and the real world. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE'10, pages 73–80, June 2010. 161

1597. C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC'13, Nov. 2013. 146

1598. J. Ruohonen and V. Leppänen. How PHP releases are adopted in the wild? In *eprint arXiv:cs.SE/1710.05570*, Oct. 2017. 94

1599. A. L. Russell. 'rough consensus and running code' and the internet-OSI standards war. *IEEE Annals of the History of Computing*, 28(3):48–61, July-Sept. 2006. 128

1600. R. Sabherwal, A. Jeyaraj, and C. Chowa. Information systems success: Individual and organizational determinants. *Management Science*, 52(12):1849–1864, Dec. 2006. 261

1601. R. Saborido, V. Arnaoudova, G. Beltrame, F. Khomh, and G. Antoniol. On the impact of sampling frequency on software energy measurements. *Peerj PrePrints*, 3:e1219, July 2015. 365, 366

1602. H. Sackman, W. J. Erikson, and E. E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM*, 11(1):3–11, Jan. 1968. 7

1603. M. Sadat, A. B. Bener, and A. V. Miranskyy. Rediscovery datasets: Connecting duplicate reports. In *eprint arXiv:cs.SE/1703.06337v1*, Mar. 2017. 147, 156, 157

1604. M. Sadinle. On the performance of dual system estimators of population size: A simulation study. Documentos de CERAC No. 13, Centro de Recursos parael Análisis de Conflictos, Bogotá, Columbia, Dec. 2008. 99

1605. D. Sahal. *Patterns of Technological Innovation*. Addison–Wesley, Dec. 1981. 3

1606. S. K. Sahoo, J. Criswell, and V. Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE'10, pages 485–494, May 2010. 148

1607. J. Sajaniemi and R. N. Prieto. Roles of variables in experts' programming knowledge. In *17th Workshop of the Psychology of Programming Interest Group*, PPIG'05, pages 145–159, June 2005. 202

1608. A. Salahirad, H. Almulla, and G. Gay. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Software Testing, Verification and Reliability*, 29(4-5):e1701, June-Aug. 2019. 169

1609. P. H. Salus. *A Quarter Century of UNIX*. Addison–Wesley, 1994. 111

1610. P. H. Salus. Duelling UNIXes and the UNIX wars. *;login:*, 40(2):66–68, Apr. 2015. 111

1611. A. Sampson, W. Dietl, E. Fortuna, and D. Gnanapragasam. EnerJ: Approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI'11, pages 164–174, June 2011. 364

1612. D. M. Sanbonmatsu, S. S. Posavac, A. A. Behrends, S. M. Moore, and B. N. Uchino. Why a confirmation strategy dominates psychological science. *PLoS ONE*, page e0138197, Sept. 2015. 23

1613. D. Sarkar. *Lattice Multivariate Data Visualization with R*. Springer Science+Business Media, 2008. 221

1614. M. Savić, M. Ivanović, Z. Budimac, and M. Radovanović. Do students' programming skills depend on programming language? In *American Institute of Physics Conference Proceedings*, page 240006, June 2016. 192

1615. SC22/WG14. *Implementation of ISO/IEC 9899:1990 (E) Programming languages – C*. British Standards Institution, Dec. 1990. 158

1616. W. Scacchi. Understanding software productivity. In W. D. Hurley, editor, *Software Engineering and Knowledge Engineering: Trends for the Next Decade Vol. 4*, chapter 10, pages 273–316. World Scientific Press, June 1995. 55

1617. S. R. Schach, T. O. S. Adeshiyan, D. Balasubramanian, G. Madl, E. P. Osses, S. Singh, K. Suwanmongkol, M. Xie, and D. G. Feitelson. Common coupling and pointer variables, with application to a Linux case study. *Software Quality Journal*, 15(1):99–113, Mar. 2006. 166

1618. S. R. Schach, B. Jin, L. Yu, G. Z. Heller, and J. Offutt. Determining the distribution of maintenance categories: Survey versus measurement. *Empirical Software Engineering*, 8(4):351–363, Dec. 2003. 351, 352

1619. J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. In *Proceedings of the VLDB Endowment*, pages 460–471, Sept. 2010. 370, 371

1620. K. W. Schaie. *Developmental Influences on Adult Intelligence: The Seattle Longitudinal Study*. Oxford University Press, second edition, 2013. 56

1621. R. R. Schaller. *Technological Innovation in the Semiconductor Industry: A Case Study of the International Technology Roadmap for Semiconductors (ITRS)*. PhD thesis, George Mason University, 2004. 91

1622. M. Schief. *Business Models in the Software Industry: The Impact on Firm and M&A Performance*. Springer Gabler, Apr. 2014. 80, 84

1623. F. L. Schmidt, I.-S. Oh, and J. A. Shaffer. The validity and utility of selection methods in personnel psychology: Practical and theoretical implications of 100 years of research findings. Working paper, Tippie College of Business, University of Iowa, Oct. 2016. 19

1624. E. Schneider, M. Maruyama, S. Dehaene, and M. Sigman. Eye gaze reveals a fast, parallel extraction of the syntax of arithmetic formulas. *Cognition*, 125(3):475–490, Dec. 2012. 27

1625. S. Schneider. The dirty little secret of software pricing. website, 2012. http://www.rti.com/whitepapers/Dirty_Little_Secret.pdf. 81

1626. J. Schock, M. J. Cortese, M. M. Khanna, and S. Toppi. Age of acquisition estimates for 3,000 disyllabic words. *Behavior and Research Methods*, 44(4):971–977, Dec. 2012. 190

1627. A. Scholey and L. Owen. Effects of chocolate on cognitive function and mood: a systematic review. *Nutrition Reviews*, 71(10):665–681, Apr. 2013. 55

1628. R. Schöne, D. Hackenberg, and D. Molka. Memory performance at reduced CPU clock speeds: An analysis of current x86_64 processors. In *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, HotPower'12, Oct. 2012. 219, 367

1629. M. Schonlau, W. DuMouchel, W.-H. Ju, A. F. Karr, M. Theus, and Y. Vardi. Computer intrusion: Detecting masquerades. *Statistical Science*, 16(1):58–74, 2001. 68

1630. L. J. Schooler and R. Hertwig. How forgetting aids heuristic inference. *Psychological Review*, 112(3):610–628, July 2005. 33

1631. E. R. Schotter, B. Angele, and K. Rayner. Parafoveal processing in reading. *Attention, Perception & Psychophysics*, 74(1):5–35, Jan. 2012. 27

1632. J.-P. Schraepler and G. G. Wagner. Identification of faked interviews in surveys by means of Benford's law?: An analysis by means of genuine fakes in the raw data of SOEP. Technical report, Technische Universiät Berlin, Aug. 2004. 382

1633. A. Schulman, M. Pietrek, and D. Maxey. *Undocumented Windows: A Programmers Guide to Reserved Microsoft Windows API Functions*. Addison–Wesley, July 1992. 113

1634. J. F. Schulz, D. Bahrami-Rad, J. P. Beauchamp, and J. Henrich. The church, intensive kinship, and global psychological variation. *Science*, 366(6466):eaau5141, Nov. 2019. 19

1635. M.-A. Schulz, B. Schmalbach, P. Brugger, and K. Witt. Analysing humanly generated random number sequences: A pattern-based approach. *PLoS ONE*, 7(7):e41531, July 2012. 382

1636. P. Schuurman, E. Berghout, and P. Powell. Benefits are from Venus, costs are from Mars. CITER WP/010/PSEBPP, University of Groningen Centre for IT Economics Research, June 2008. 115

1637. E. S. Schwartz and C. Zozaya-Gorostiza. Investment under uncertainty in information technology: Acquisition and development projects. *Management Science*, 49(1):57–70, Jan. 2003. 62

1638. C. Scott. Numbers every programmer should know. website, Oct. 2016. https://github.com/colin-scott/interactive_latencies. 226, 227

1639. C. F. Scott, P. Cole, R. B. Hesse, and P. R. Malone. UNITED STATES OF AMERICA, et al., v. ORACLE CORPORATION. Plaintiff's post-trial brief CASE NO. C 04-0807 VRW, UNITED STATES DISTRICT COURT NORTHERN DISTRICT OF CALIFORNIA SAN FRANCISCO DIVISION, July 2004. 98

1640. M. D. Scott. Tort liability for vendors of insecure software: Has the time finally come? *Maryland Law Review*, 67(2):425–484, 2008. 150

1641. P. D. Scott and M. Fasli. Benford's law: An empirical investigation and a novel explanation. CSM Technical Report 349, Department of Computer Science, University of Essex, Aug. 2001. 382

1642. Intel overstates FPU accuracy. Personal website, June 2013. http://notabs.org/fpuaccuracy. 146

1643. S. Scribner. Modes of thinking and ways of speaking: culture and logic reconsidered. In P. N. Johnson-Laird and P. C. Wason, editors, *Thinking: Readings in Cognitive Science*, chapter 29, pages 483–500. Cambridge University Press, 1977. 41

1644. R. C. Seacord. *The CERT C Secure Coding Standard*. Addison–Wesley, 2009. 179

1645. R. C. Seamans, Jr. *Aiming at Targets: The Autobiography of Robert C. Seamans, Jr.* NASA History Office, 1996. 116

1646. SEC. The world's largest hedge fund is a fraud. SEC MADOFF EXHIBITS-04451, Nov. 2005. November 7, 2005 Submission to the SEC, Madoff Investment Securities, LLC. 382

1647. P. Sehgal, V. Tarasov, and E. Zadok. Evaluating performance and energy in file system server workloads. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, Feb. 2010. 369

1648. J. Selby and K. Mayer. Startup firm acquisitions as a human resource strategy for innovation: The acqhire phenomenon. *Academy of Management Annual Meeting Proceedings*, 1:17109–17109, Nov. 2013. 105

1649. R. W. Selby, Jr., V. R. Basili, and F. T. Baker. CLEANROOM software development: An empirical evaluation. Technical Report TR-1415, Department of Computer Science, University of Maryland, Feb. 1985. 126

1650. L. L. Selwyn. *Economies of Scale in Computer Use: Initial Tests and Implications for the Computer Utility*. PhD thesis, Alfred P. Sloan School of Management, June 1969. 101

1651. J. A. Sexton. Detecting errors in software using a parameter checker: An analysis. Thesis (m.s.), Rochester Institute of Technology, Apr. 1989. 159

1652. N. Shadbolt. Shadbolt review of computer sciences degree accreditation and graduate employability. Technical Report IND/16/5, Department for Business, Innovation & Skills, UK, Apr. 2016. 68, 357

1653. T. M. Shaft and I. Vessey. The relevance of application domain knowledge: Characterizing the computer program comprehension process. *Journal of Management Information Systems*, 15(1):51–78, 1998. 182

1654. C. R. Shaliz. g, a statistical myth. blog: Three-Toed Sloth, Oct. 2007. http://bactra.org/weblog/523.html. 50

1655. J. Shallit. Randomized algorithms in "primitive" cultures or what is the oracle complexity of a dead chicken? *ACM SIGACT News*, 23(4):77–80, Sept.-Nov. 1992. 182

1656. C. Shaoul, R. H. Baayen, and C. F. Westbury. N-gram probability effects in a cloze task. *The Mental Lexicon*, 9(3):437–472, 2014. 190

1657. C. Shapiro and H. R. Varian. The art of standards wars. *California Management Review*, 41(2):8–32, Jan. 1999. 77

1658. R. Sharp, M. Paul, A. Nagesh, D. Bell, and M. Surdeanu. Grounding gradable adjectives through crowdsourcing. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation*, LREC 2018, May 2018. 149

1659. W. F. Sharpe. *The Economics of Computers*. Columbia University Press, 1969. 101

1660. O. Shatnawi. Measuring commercial software operational reliability: an interdisciplinary modelling approach. *Eksploatacja i Niezawodnosc – Maintenance and Reliability*, 16(4):585–594, 2014. 151

1661. D. E. Shaw, R. O. Dror, J. K. Salmon, J. P. Grossman, K. M. Mackenzie, J. A. Bank, C. Young, M. M. Deneroff, B. Batson, K. J. Bowers, E. Chow, M. P. Eastwood, D. J. Ierardi, J. L. Klepeis, J. S. Kuskin, R. H. Larson, K. Lindorff-Larsen, P. Maragakis, M. A. Moraes, S. Piana, Y. Shan, and B. Towles. Millisecond-scale molecular dynamics simulations on Anton. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC'09, page 39, Nov. 2009. 108

1662. B. R. Shear and B. D. Zumbo. False positives in multiple regression: Unanticipated consequences of measurement error in the predictor variables. *Educational and Psychological Measurement*, 73(5):733–756, Oct. 2013. 284

1663. D. Shefer. Pricing for software product managers. ???, 2005. 82

1664. B. A. Sheil. The psychological study of programming. *ACM Computing Surveys*, 13(1):101–120, Mar. 1981. 6

1665. S. Shekhar, M. Dietz, and D. S. Wallach. AdSplit: Separating smartphone advertising from applications. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, page 28, Aug. 2012. 85

1666. T.-J. Shen, A. Chao, and C.-F. Lin. Predicting the number of new species in further taxonomic sampling. *Ecology*, 84(3):798–804, Mar. 2003. 100

1667. V. Y. Shen, S. D. Conte, and H. E. Dunsmore. Software science revisited: A critical analysis of the theory and its empirical support. Technical Report CSD-TR 375, Purdue University, Jan. 1981. 194

1668. A. Shenhav, S. Musslick, F. Lieder, W. Kool, T. L. Griffiths, J. D. Cohen, and M. M. Botvinick. Toward a rational and mechanistic account of mental effort. *Annual Review of Neuroscience*, 40:99–124, July 2017. 24

1669. R. N. Shepard, C. I. Hovland, and H. M. Jenkins. Learning and memorization of classifications. *Psychological Monographs: General and Applied*, 75(15):1–39, 1961. 39, 40

1670. R. N. Shepard and J. Metzler. Mental rotation of three-dimensional objects. *Science*, 171:701–703, Feb. 1971. 20

1671. S. B. Sheppard and E. Kruesi. The effects of the symbology and spatial arrangement of software specifications in a coding task. Technical Report TR-81-388200-3, Information Systems Programs, General Electric, Feb. 1981. 158

1672. M. Shepperd, C. Mair, and M. Jørgensen. An experimental evaluation of a de-biasing intervention for professional software developers. In *eprint arXiv:cs.SE/1804.03919*, Apr. 2018. 123

1673. L. Shi, H. Zhong, T. Xie, and M. Li. An empirical study on evolution of API documentation. In *Proceedings of the 14th international conference on Fundamental approaches to software engineering*, FASE'11/ETAPS'11, pages 416–431, Apr. 2011. 114

1674. E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K. ichi Matsumoto. Predicting re-opened bugs: A case study on the Eclipse project. In *17th Working Conference on Reverse Engineering*, WCRE'10, pages 249–258, Oct. 2010. 346, 347

1675. E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan. Understanding the impact of code and process metrics on post-release defects: A case study on the Eclipse project. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM'10, pages 1–4, Sept. 2010. 160

1676. M. Shimasaki, S. Fukaya, K. Ikeda, and T. Kiyono. An analysis of Pascal programs in compiler writing. *Software–Practice and Experience*, 10(2):149–157, Feb. 1980. 125, 126

1677. A. L. Shimpi and B. Klug. They're (almost) all dirty: The state of cheating in Android benchmarks. website, Oct. 2013. http://www.anandtech.com/show/7384/state-of-cheating-in-android-benchmarks. 362

1678. T. C. Shrum. Calibration and validation of the checkpoint model to the air force electronic systems center software database. Thesis (m.s.), Graduate School of Logistics and Acquisition Management or the Air Force Institute of Technology, USA, Sept. 1997. 6

1679. A. Shterenlikht. On quality of implementation of Fortran 2008 complex intrinsic functions on branch cuts. In *eprint arXiv:cs.MS/1712.10230*, Dec. 2017. 161

1680. O. Shy. *How to Price: A Guide to Pricing Techniques and Yield Management*. Cambridge University Press, 2008. 81

1681. R. M. Siegfried, J. P. Siegfried, and G. Alexandro. A longitudinal analysis of the Reid list of first programming languages. *Information Systems Education Journal*, 14(6):47–54, Nov. 2016. 111

1682. N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information and Software Technology*, 55(3):491–507, Mar. 2013. 134

1683. I. Siket, Á. Beszédes, and J. Taylor. Differences in the definition and calculation of the LOC metric in free tools. Technical Report TR2014-001, Department of Software Engineering, University of Szeged, 2014. 253

1684. R. Silberzahn, E. L. Uhlmann, D. P. Martin, P. Anselmi, F. Aust, E. Awtrey, Š. Bahník, F. Bai, C. Bannard, E. Bonnier, R. Carlsson, F. Cheung, G. Christensen, R. Clay, M. A. Craig, A. D. Rosa, L. Dam, M. H. Evans, I. F. Cervantes, N. Fong, M. Gamez-Djokic, A. Glenz, S. Gordon-McKeon, T. J. Heaton, K. Hederos, M. Heene, A. J. H. Mohr, F. Högden, K. Hui, M. Johannesson, J. Kalodimos, E. Kaszubowski, D. M. Kennedy, R. Lei, T. A. Lindsay, S. Liverani, C. R. Madan, D. Molden, E. Molleman, R. D. Morey, L. B. Mulder, B. R. Nijstad, N. G. Pope, B. Pope, J. M. Prenoveau, F. Rink, E. Robusto, H. Roderique, A. Sandberg, E. Schlüter, F. D. Schönbrodt, M. F. Sherman, S. A. Sommer, K. Sotak, S. Spain, C. Spörlein, T. Stafford, L. Stefanutti, S. Tauber, J. Ullrich, M. Vianello, E.-J. Wagenmakers, M. Witkowiak, S. Yoon, and B. A. Nosek. Many analysts, one data set: Making transparent how variations in analytic choices affect results. *Advances in Methods and Practices in Psychological Science*, 1(3):337–356, Apr. 2018. 249

1685. T. Simcoe. Standard setting committees: Consensus governance for shared technology platforms. *American Economic Review*, 102(1):305–336, Feb. 2013. 77

1686. T. Simcoe. Modularity and the evolution of the internet. In A. Goldfarb, S. M. Greenstein, and C. E. Tucker, editors, *Economic Analysis of the Digital Economy*, chapter 1, pages 21–47. University of Chicago Press, May 2015. 179, 180

1687. T. S. Simcoe and D. M. Waguespack. Status, quality and attention: What's in a (missing) name? *Management Science*, 57(2):274–290, Sept. 2011. 70, 71

1688. K. M. Simmons and D. Sutter. False alarms, tornado warnings, and tornado casualties. *Weather, Climate, and Society*, 1(1):38–53, Oct. 2009. 229

1689. H. A. Simon. *Models of Bounded Rationality: Behavioral Economics and Business Organization*. The MIT Press, 1982. 51

1690. H. A. Simon. Making management decisions: the role of intuition and emotion. *The Academy of Management Executive (1987-1989)*, 1(1):57–64, Feb. 1987. 35

1691. I. Simonson. Choice based on reasons: The case of attraction and compromise effects. *Journal of Consumer Research*, 16:158–173, Sept. 1989. 51

1692. I. C. Simpson, P. Mousikou, J. M. Montoya, and S. Defior. A letter visual-similarity matrix for Latin-based alphabets. *Behavior and Research Methods*, 45(2):431–439, June 2013. 21

1693. J. Singer, M. Luján, and I. Watson. Meaningful type names as a basis for object lifetime prediction. In *Proceedings of the 2008 ACM SIGPLAN international conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA'08, page ???, Apr. 2008. 198

1694. P. V. Singh and C. Phelps. Networks, social influence, and the choice among competing innovations: Insights from open source software licenses. *Information Systems Research*, 24(3):539–560, Nov. 2013. 65

1695. D. I. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanović, E. F. Koren, and M. Vokác. Conducting realistic experiments in software engineering. In *Proceedings of the 2002 International Symposium on Empirical Software Engineering*, ISESE'02, pages 17–26, Oct. 2002. 356

1696. D. I. K. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanović, N.-K. Liborg, and A. C. Rekdal. A survey of controlled experiments in software engineering. Technical Report 2004-4, SIMULA Research Laboratory, 2004. 354

1697. D. Skau and R. Kosara. Arcs, angles, or areas: Individual data encodings in pie and donut charts. In *Eurographics Conference on Visualization*, EuroVis'16, pages 121–130, June 2016. 221

1698. J. Skelley. Open source tactics: Bargaining power for strategic litigation. *Chicago-Kent Journal of Intellectual Property*, 16(1), 2016. 66

1699. I. Skoulis. Analysis of schema evolution for databases in open-source software. Thesis (m.s.), University of Ioannina, Greece, Sept. 2013. 141, 142

1700. G. Slade. *Made to Break: Technology and Obsolescence in America*. Harvard University Press, 2007. 4, 163

1701. S. A. Slaughter, S. Ang, and W. F. Boh. Firm-specific human capital and compensation-organizational tenure profiles: An archival analysis of salary data for IT professionals. *Human Resource Management*, 46(3):373–394, 2007. 67

1702. S. A. Sloman. The empirical case for two systems of reasoning. *Psychological Bulletin*, 119(1):3–22, 1996. 41

1703. S. A. Sloman. Categorical inference is not a tree: The myth of inheritance hierarchies. *Cognitive Psychology*, 35(1):1–33, Feb. 1998. 39

1704. S. A. Sloman, M. C. Harrison, and B. C. Malt. Recent exposure affects artifact naming. *Memory & Cognition*, 30(5):687–695, 2002. 190

1705. S. A. Sloman and D. Lagnado. Causality in thought. *Annual Review of Psychology*, 66:223–247, 2015. 45

1706. S. A. Sloman and D. A. Lagnado. Do we "do"? *Cognitive Science*, 29(1):5–39, Jan.-Feb. 2005. 45

1707. P. Slovic. *The Perception of Risk*. Earthscan Publications Ltd, 2000. 148

1708. P. E. Smaldino and R. McElreath. The natural selection of bad science. *Royal Society Open Science*, 3:160384, Aug. 2016. 9

1709. G. K. Smith, A. A. Barbour, T. L. McNaugher, M. D. Rich, and W. L. Stanley. The use of prototypes in weapon system development. Report R-2345-AF, The RAND Corporation, Mar. 1981. 131

1710. C. M. So. An analysis of mathematical expressions used in practice. Thesis (m.s.), The University of Western Ontario, 2005. 195

1711. F. Söhnchen and S. Albers. Pipeline management for the acquisition of industrial projects. *Industrial Marketing Management*, 39(8):1356–1364, Nov. 2010. 119

1712. M. Sojer, O. Alexy, S. Kleinknecht, and J. Henkel. Understanding the drivers of unethical programming behavior: The inappropriate reuse of internet-accessible code. *Journal of Management Information Systems*, 31(3):287–325, 2014. 120

1713. Solganick & Co. Software M&A update. http://www.solganickco.com/wp-content/uploads/2017/02/Solganick-Software-Q4-2016-final.pdf, Apr. 2016. 89

1714. M. B. Solomon, Jr. Economies of scale and the IBM System/360. *Communications of the ACM*, 9(6):435–440, June 1966. 90

1715. G. S. Sommer. *Astronomical Odds A Policy Framework for the Cosmic Impact Hazard*. PhD thesis, Pardee RAND Graduate School, USA, June 2004. 148

1716. J. Sonnemans. Price clustering and natural resistance points in the Dutch stock market: A natural experiment. *European Economic Review*, 50(8):1937–1950, Nov. 2006. 47

1717. C. Soto-Valero, M. Monperrus, N. Harrand, and B. Baudry. A comprehensive study of bloated dependencies in the Maven ecosystem. In *eprint arXiv:cs.SE/2001.07808*, Jan. 2020. 194

1718. R. W. Soukoreff. *Quantifying Text Entry Performance*. PhD thesis, York University, Toronto, Canada, Apr. 2010. 21

1719. SPEC. SPEC power_ssj 2008. http://spec.org/power_ssj2008, June 2016. 310

1720. SPEC. Standard performance evaluation corporation. http://spec.org, Sept. 2020. 87, 213, 215, 265, 303, 362

1721. SPECpower Committee. Power and performance benchmark methodology. V 2.2, Standard Performance Evaluation Corporation (SPEC), Dec. 2014. 365

1722. I. Spence. Visual psychophysics of simple graphical elements. *Journal of Experimental Psychology: Human Perception and Performance*, 16(4):683–692, Nov. 1990. 221

1723. I. Spence and S. Lewandowsky. Displaying proportions and percentages. *Applied Cognitive Psychology*, 5(1):61–77, Apr. 1991. 220, 221

1724. M. Spence. Job market signalling. *The Quarterly Journal of Economics*, 87(3):355–374, Aug. 1973. 68

1725. D. Sperber and D. Wilson. *Relevance: Communication and Cognition*. Blackwell Publishers, second edition, 1995. 41, 174

1726. D. Spinellis. *Code Reading: The Open Source Perspective*. Addison–Wesley, 2003. 179

1727. D. Spinellis, V. Karakoidas, and P. Louridas. Comparative language fuzz testing: Programming languages vs. fat fingers. In *Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU 2012, pages 25–34, Oct. 2012. 159, 160

1728. D. Spinellis, Z. Kotti, K. Kravvaritis, G. Theodorou, and P. Louridas. A dataset of enterprise-driven open source software. In *eprint arXiv:cs.SE/2002.03927*, Feb. 2020. 11

1729. J. Spolsky. Fog Creek professional ladder. https://www.joelonsoftware.com/2009/02/13/fog-creek-professional-ladder, Feb. 2009. 66

1730. J. Sprouse and D. Almeida. Assessing the reliability of textbook data in syntax: Adger's core syntax. *Journal of Linguistics*, 48(3):609–652, Nov. 2012. 158

1731. D. Spuler. *C++ and C debugging, testing and reliability*. Prentice-Hall, Inc, 1994. 179

1732. L. R. Squire and A. J. O. Dede. Conscious and unconscious memory systems. *Perspectives in Biology*, 7(3):a021667, Mar. 2015. 27

1733. J. Srinivasan. *Lifetime Reliability Aware Microprocessor*. PhD thesis, University of Illinois at Urbana-Champaign, Oct. 2006. 161

1734. E. B. Staats. Millions in savings possible in converting programs from one computer to another. Technical Report FGMSD-77-34, Office of Management and Budget, Nationai Bureau of Standards, Sept. 1977. 109

1735. C. B. Stabell and Ø. D. Fjeldstad. Configuring value for competitive advantage: On chains, shops, and networks. *Strategic Management Journal*, 19(5):413–437, May 1998. 80, 81

1736. Standish Group. The CHAOS report. Technical report, The Standish Group International, Inc, Aug. 1994. 118

1737. P. Stanley-Marbell, V. Estellers, and M. Rinard. Crayon: Saving power through shape and color approximation on next-generation displays. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys'16, page 11, Apr. 2016. 364

1738. K. E. Stanovich. *Who Is Rational? Studies of Individual Differences in Reasoning*. Lawrence Erlbaum Associates, 1999. 41, 42

1739. K. E. Stanovich and R. F. West. Individual differences in reasoning: Implications for the rationality debate? *Behavioral and Brain Sciences*, 23(5):645–726, Oct. 2000. 41

1740. M. Staples, R. Kolanski, G. Klein, C. Lewis, J. Andronick, T. Murray, R. Jeffery, and L. Bass. Formal specifications better than function points for code sizing. In *International Conference on Software Engineering*, ICSE'13, pages 1257–1260, May 2013. 212

1741. J. Starek. A large-scale analysis of Java API usage. Thesis (m.s.), Institut für Informatik, Universität Koblenz-Landau, Mar. 2010. 298

1742. E. Starr. The use, abuse, and enforceability of non-compete and no-poach agreements: A brief review of the theory, evidence, and recent reform efforts. Issue brief, Economic Innovation Group, Washington D.C., Feb. 2019. 105

1743. T. N. Starr, L. K. Picton, and J. W. Thornton. Alternative evolutionary histories in the sequence space of an ancient protein. *Nature*, 549:409–413, Sept. 2017. 92

1744. M. Stasinopoulos, B. Rigby, V. Voudouris, G. Heller, and F. D. Bastiani. Flexible regression and smoothing: The GAMLSS packages in R. draft book, July 2015. 298

1745. G. Stasser and W. Titus. Effects of information load and percentage of shared information on the dissemination of unshared information during group discussion. *Journal of Personality and Social Psychologs*, 53(1):81–93, July 1987. 76

1746. M. Steele and J. Chaseling. Powers of discrete goodness-of-fit test statistics for a uniform null against a selection of alternative distributions. *Communications in Statistics-Simulation and Computation*, 35(4):1067–1075, Apr. 2006. 238

1747. G. L. Steele, Jr. and R. P. Gabriel. The evolution of Lisp. In *The second ACM SIGPLAN conference on History of Programming Languages*, HOPL II, pages 231–270, Apr. 1993. 109

1748. R. G. Steen, A. Casadevall, and F. C. Fang. Why has the number of scientific retractions increased? *PLoS ONE*, 8(7), Apr. 2013. 9

1749. J. Steffens. *Newgames: Strategic Competition in the PC revolution*. Pergamon Press, 1994. 90

1750. T. Stengos and E. Zacharias. Intertemporal pricing and price discrimination: A semiparametric hedonic analysis of the personal computer market. Discussion Paper 2002-11, Department of Economics, University of Cyprus, June 2002. 83

1751. K. Stenning and M. van Lambalgen. Semantics as a foundation for psychology: A case study of Wason's selection task. *Journal of Logic, Language and Information*, 10(3):273–317, June 2001. 41

1752. K. Stenning and M. van Lambalgen. A little logic goes a long way: basing experiment on semantic theory in the cognitive science of conditional reasoning. *Cognitive Science*, 28(4):481–530, July-Aug. 2004. 42

1753. K. Stenning and M. van Lambalgen. *Human Reasoning and Cognitive Science*. MIT Press, 2008. 41

1754. M. A. Stephens. EDF statistics for goodness of fit and some comparisons. *Journal of the American Statistical Association*, 69(347):730–737, Sept. 1974. 236

1755. R. J. Sternberg and E. M. Weil. An aptitude-strategy interaction in linear syllogistic reasoning. Technical Report 15, Department of Psychology, Yale University, Apr. 1979. 42

1756. S. Sternberg. Memory-scanning: Mental processes revealed by reaction-time experiments. *American Scientist*, 57(4):421–457, 1969. 29

1757. A. Stevens and P. Coupe. Distortions in judged spatial relations. *Cognitive Psychology*, 10(4):422–437, Oct. 1978. 39

1758. N. Stewart, N. Chater, and G. D. A. Brown. Decision by sampling. *Cognitive Psychology*, 53(1):1–26, Jan. 2006. 51

1759. N. Stewart, C. Ungemach, A. J. L. Harris, D. M. Bartels, B. R. Newell, G. Paolacci, and J. Chandler. The average laboratory samples a population of 7,300 Amazon Mechanical Turk workers. *Judgment and Decision Making*, 10(5):479–491, Sept. 2015. 357

1760. G. Stikkel. Dynamic model for the system testing process. *Information and Software Technology*, 48(7):578–585, July 2006. 166

1761. V. Stodden, P. Guo, and Z. Ma. Toward reproducible computational research: An empirical analysis of data and code policy adoption by journals. *PLoS ONE*, 8(6):e13636, June 2013. 9

1762. Z. Stojanova, D. Dobrilovic, and J. Stojanova. Analyzing trends for maintenance request process assessment: Empirical investigation in a very small software company. *Theory and Applications of Mathematics & Computer Science*, 3(2):59–74, Nov. 2013. 140

1763. G. P. Stone, D. B. Levin, H. Hwang, M. Kim, and C. Mckay. JANET SKOLD and DAVID DOSSANTOS, on behalf of themselves and all others similarly situated, v. INTEL CORPORATION, HEWLETT PACKARD COMPANY and DOES 1-50, case no. 1-05-CV-039231, filing #g-64475. Opinion, Superior court of the state of California for the county of Santa Clara, 2014. 362

1764. H. S. Stone. Life-cycle cost analysis of instruction-set architecture standardization for military computer systems. *Computer*, 12(4):35–47, Apr. 1979. 92

1765. J. Stone, M. Greenwald, C. Partridge, and J. Hughes. Performance of checksums and CRCs over real data. *IEEE/ACM Transactions on Networking*, 6(5):529–543, Oct. 1998. 147

1766. P. Stoneman. *Technological Diffusion and the Computer Revolution: The UK experience*. Cambridge University Press, Jan. 1976. 1, 82, 88

1767. D. Straker. *C-Style standards and guidelines*. Prentice-Hall, Inc, 1992. 179

1768. S. Strand, I. J. Deary, and P. Smith. Sex differences in cognitive abilities test scores: A UK national picture. *British Journal of Educational Psychology*, 76(3):463–480, Apr. 2006. 18, 19

1769. W. Stroebe, B. A. Nijstad, and E. F. Rietzschel. Beyond productivity loss in brainstorming groups: The evolution of a question. Working Paper No. 2014-05, Center for Research in Economics, Management and the Arts, CREMA Südstrasse 11 CH, 2014. 76

1770. H. . Strong. Mozilla foundation and subsidiary december 31, 2015 and 2014. Independent auditors' report and consolidated financial statements, Hood & Strong LLC, Nov. 2016. 117

1771. R. Sudan, S. Ayers, P. Dongier, A. Muente-Kunigami, and C. Z.-W. Qiang. The global opportunity in IT-based services: Assessing and enhancing country competitiveness. Report, The World Bank, 2010. 58

1772. C. Sun, V. Le, Q. Zhang, and Z. Su. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA'16, pages 294–305, July 2016. 156, 157

1773. L. Sun. What we are paying for: A quality adjusted price index for laptop microprocessors. Honors thesis, Wellesley College, Apr. 2014. 81, 82

1774. Sun Microsystems, Inc. Java code conventions. Technical report, Sun Microsystems, Inc, Sept. 1997. 205, 351

1775. T. Sunada, A. Monden, and K. Matsumoto. On estimating source lines of code from a binary program. In *Joint Conference of International Workshop on Software Measurement and International Conference on Software Process and Product Measurement*, IWSM/Mensura 2011, pages 3–6, Nov. 2011. 176

1776. C. R. Sunstein. *The Cost-Benefit Revolution*. The MIT Press, Aug. 2018. 145

1777. A. Suresh, B. N. Swamy, E. Rohou, and A. Seznec. Intercepting functions for memoization: A case study using transcendental functions. *Transactions on Architecture and Code Optimization*, 12(2):18, July 2015. 198

1778. P. Suthipornopas, P. Leelaprute, A. Monden, H. Uwano, Y. Kamei, N. Ubayashi, K. Araki, K. Yamada, and K. ichi Matsumoto. Industry application of software development task measurement system: TaskPit. *IEICE Transactions on Information & Systems*, E100(3):462–472, Mar. 2017. 133

1779. K. Suzuki and S. Swanson. A survey of trends in non-volatile memory technologies: 2000-2014. In *IEEE International Memory Workshop*, IMW, pages 1–4, May 2015. 366

1780. T. N. Suzuki, D. Wheatcroft, and M. Griesser. Experimental evidence for compositional syntax in bird calls. *Nature Communications*, 7(10986), Mar. 2016. 18

1781. M. Swan and B. Smith. *Learner English: A teacher's guide to interference and other problems*. Cambridge University Press, second edition, 2001. 191

1782. E. B. Swanson and C. M. Beath. *Maintaining Information Systems in Organizations*. John Wiley & Sons, Inc, 1989. 104, 138

1783. G. M. Swift and S. M. Guertin. In-flight observations of multiple-bit upset in DRAMs. *IEEE Transactions on Nuclear Science*, 47(6):2386–2391, Dec. 2000. 162

1784. R. A. Syed, B. Robinson, and L. Williams. Does hardware configuration and processor load impact software fault observability? In *Third International Conference on Software Testing, Verification and Validation*, ICST'10, pages 285–294, Apr. 2010. 254, 255

1785. I. H. Tabernero. *A statistical examination of the properties and evolution of libre software*. PhD thesis, Universidad Rey Juan Carlos, Oct. 2008. 176, 279, 321, 330

1786. N. Taerat, N. Naksinehaboon, C. Chandler, J. Elliott, C. B. Leangsuksun, G. Ostrouchov, S. L. Scott, and C. Englemann. Blue Gene/L log analysis and time to interrupt estimation. In *International Conference on Availability, Reliability and Security*, ARES'09, pages 173–180, Oct. 2009. 381

1787. L. Takeyama. The shareware industry: Some stylized facts and estimates of rates of return. *Economics of Innovation and New Technology*, 3(2):161–174, Jan. 1994. 81

1788. P. P. Tallon, R. J. Kauffman, H. C. Lucas, A. B. Whinston, and K. Zhu. Using real options analysis for evaluating uncertain investments in information technology: Insights from the ICIS 2001 debate. *Communications of the Association for Information Systems*, 9:136–167, Sept. 2002. 115

1789. K. Y. Tam. Capital budgeting in information systems development. *Information & Management*, 23(6):345–357, Dec. 1992. 58

1790. T. Tamai. Experiment on coordination within software development teams. *Information and Software Technology*, 34(7):437–442, July 1992. 133

1791. T. Tamai and Y. Torimitsu. Software lifetime and its evolution process over generations. In *Proceedings of 1992 Conference on Software Maintenance*, pages 63–69, Nov. 1992. 61, 95, 96

1792. P.-N. Tan and V. K. J. Srivastava. Selecting the right objective measure for association analysis. *Information Systems*, 29(4):293–313, June 2004. 275

1793. E. Tang, E. Barr, X. Li, and Z. Su. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA'10, pages 131–142, July 2010. 146

1794. V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok. Generating realistic datasets for deduplication analysis. In *Proceedings of the 2012 USENIX Annual Technical Conference*, ATC'12, June 2012. 244, 358

1795. R. C. Tausworthe. Staffing implications of software productivity models. In E. C. Posner, editor, *The Telecommunication and Data Acquisition Report 42-72*, pages 70–77. Jet Propulsion Laboratory, California Institute of Technology, Oct.-Dec. 1982. 138

1796. F. W. Taylor. *The Principles of Scientific Management*. Harper & Brothers Publishers, 1919. 69

1797. Q. C. Taylor. Analysis and characterization of author contribution patterns in open source software development. Thesis (m.s.), Brigham Young University, Apr. 2012. 182

1798. M. Tedre. Computing as a science: A survey of competing viewpoints. *Minds & Machines*, 21(3):361–387, Aug. 2011. 5

1799. J. J. Tehrani. The phylogeny of little red riding hood. *PLoS ONE*, 8(11):e78871, Nov. 2013. 180

1800. J. Teixeira, G. Robles, and J. M. González-Barahona. Lessons learned from applying social network analysis on an industrial free/libre/open source software ecosystem. *Journal of Internet Services and Applications*, 6(1):1–27, 2015. 77

1801. M. Templ, B. Meindl, and A. Kowarik. Introduction to statistical disclosure control (SDC). Technical report, International Household Survey Network, Oct. 2015. 374

1802. K. Tentori, D. Osherson, L. Hasher, and C. May. Wisdom and ageing: Irrational preferences in college students but not older adults. *Cognition*, 81(3):B87–B99, 2001. 51

1803. P. E. Tetlock. Accountability: The neglected social context of judgment and choice. *Research in Organizational Behavior*, 7:297–332, 1985. 51

1804. P. E. Tetlock. An alternative metaphor in the study of judgment and choice: People as politicians. *Theory and Psychology*, 1(4):451–475, 1991. 51

1805. Tezzaron Semiconductor. Soft errors in electronic memory. Technical Report 1.1, Tezzaron Semiconductor, Naperville, IL, Jan. 2004. 162

1806. T. A. Thayer, M. Lipow, and E. C. Nelson. *Software Reliability*. North-Holland Publishing Company, 1978. 6, 147

1807. The Commission. Report of investigation pursuant to section 21(a) of the securities exchange Act of 1934: The DAO. Release No. 81207, Securities and Exchange Commission, July 2017. 144

1808. E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel. Practical performance models for complex, popular applications. In *Performance Evaluation Review*, SIGMETRICS'10, pages 1–12, June 2010. 219

1809. D. R. Thomas. *Security metrics for computer systems*. PhD thesis, Cambridge Computer Laboratory, University of Cambridge, Sept. 2015. 145

1810. M. Thomas and V. Morwitz. Penny wise and pound foolish: The left-digit effect in price cognition. *Journal of Consumer Research*, 32(1):54–64, June 2005. 82

1811. M. Thomas, D. H. Simon, and V. Kadiyali. Do consumers perceive precise prices to be lower than round prices? Evidence from laboratory and market data. Research Paper Series #09-07, Johnson School, Cornell University, Sept. 2007. 82

1812. B. Thompson. The bill gates line. blog: Stratechery, May 2018. `https://stratechery.com/2018/the-bill-gates-line`. 106

1813. P. Thompson. How much did the Liberty shipbuilders forget? *Management Science*, 53(6):908–918, June 2007. 72

1814. S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, Feb. 2010. 7, 354

1815. J. D. Tinder. Entry granting reasserted motion to dismiss (docket no. 34): Daniel wallace, v. free software foundation, inc. Case 1:05-cv-0618-JDT-TAB, UNITED STATES DISTRICT COURT SOUTHERN DISTRICT OF INDIANA INDIANAPOLIS DIVISION, Mar. 2006. 66

1816. M. A. Tinker. The relative legibility of the letters, the digits, and of certain mathematical signs. *Journal of Generative Psychology*, 1:472–494, 1928. 191

1817. B. Tognazzini. Principles, techniques, and ethics of stage magic and their application to human interface design. In *Conference on Human Factors in Computing Systems*, INTERCHI'93, pages 355–362, May 1993. 5

1818. J. E. Tomayko. Computers in spaceflight: The NASA experience. NASA Contractor Report 182505, Wichita State University, Kansas, Mar. 1988. 110

1819. J. T. Townsend. Theoretical analysis of an alphabetic confusion matrix. *Perception & Psychophysics*, 9(1A):40–50, 1971. 21

1820. T. S. Traaen. The Brooks Act: An 8-bit act in a 64-bit world? An investigation of the Brooks Act and its implications to the department of defense information technology acquisition process. Executive Research Project S18, The Industrial College of the Armed Forces, National Defense University, Washington, D.C., May 1995. 101

1821. Transport, Department for. The accidents sub-objective. Transport Analysis Guidance Unit 3.4.1, Department for Transport, United Kingdom, Apr. 2011. 148

1822. L. M. Trick and Z. W. Pylyshyn. What enumeration studies can show us about spatial attention: Evidence for limited capacity preattentive processing. *Journal of Experimental Psychology: Human Perception and Performance*, 19(2):331–351, 1993. 46

1823. J. E. Triplett. Performance measures for computers. In *Deconstructing the Computer*, pages 99–139, Feb. 2003. 1

1824. D. Trippas, D. Kellen, H. Singmann, G. Pennycook, D. J. Koehler, J. A. Fugelsang, and C. Dubé. Characterizing belief bias in syllogistic reasoning: A hierarchical Bayesian meta-analysis of ROC data. *Psychonomic Bulletin and Review*, 25(2):2141–2174, Apr. 2018. 43

1825. K. S. Trivedi. *Probability & Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley & Sons, Inc, second edition, 2002. 244

1826. J. S. Trueblood and J. R. Busemeyer. A quantum probability account of order effects in inference. *Cognitive Science*, 35(8):1518–1552, Nov.-Dec. 2011. 36

1827. C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter. A study of modern Linux API usage and compatibility: What to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys'16, page 16, Apr. 2016. 112

1828. N. P. Tschacher. Typosquatting in programming language package managers. Thesis (b.sc.), Department of Informatics, University of Hamburg, Mar. 2016. 161

1829. T. K. Tsingos. Enforceability of free/open source software licensing terms: A critical review of the global case - law. In *Fourth International Conference on Information Law*, ICIL 2011, May 2011. 66

1830. TSMC. TSMC historical operating data. http://www.tsmc.com/english/investorRelations/historical_information.htm, May 2017. 91

1831. M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4):e1838, Apr. 2017. 136

1832. T. S. Tullis and J. N. Stetson. A comparison of questionnaires for assessing website usability. In *Proceedings of Usability Professionals Association*, pages 1–12, June 2004. 372

1833. J. Turley. Embedded processors. http://www.extremetech.com, Jan. 2002. 90, 295

1834. H. Turner and D. Firth. Bradley-Terry models in R: The BradleyTerry2 package. *Journal of Statistical Software*, 48(9):1–21, 2012. 350

1835. H. Turner and D. Firth. *Generalized nonlinear models in R: An overview of the gnm package*. University of Warwick, UK, 1.0-8 edition, Apr. 2015. 314

1836. M. L. Turner and R. W. Engle. Is working memory capacity task dependent? *Journal of Memory and Language*, 28(2):127–154, Apr. 1989. 186

1837. R. Turner. *Weathering Heights: The Emergence of Aeronautical Meteorology as an Infrastructural Science*. PhD thesis, University of Pennsylvania, May 2010. 2, 88

1838. L. D. Tyson. *Who's Bashing Whom? Trade Conflict in High-Technology Industries*. Institute for International Economics, Nov. 1992. 57

1839. J. Tzelgov, V. Yehene, L. Kotler, and A. Alon. Automatic comparisons of artificial digits never compared: Learning linear ordering relations. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 26(1):103–120, 2000. 48

1840. UBM. Then, now: What's next? Embedded Market Study 2014, UBM Electronics, 2014. 110

1841. A. Čaušević, R. Shukla, S. Punnekkat, and D. Sundmark. Effects of negative testing on TDD: An industrial experiment. In H. Baumeister and B. Weber, editors, *Agile Processes in Software Engineering and Extreme Programming*, volume 149 of *Lecture Notes in Business Information Processing*, pages 91–105. Springer Berlin Heidelberg, 2013. 170

1842. The ultimate Debian database. website, 2014. http://wiki.debian.org/UltimateDebianDatabase. 146, 217, 293

1843. G. Ülkümen, M. Thomas, and V. G. Morwitz. Will i spend more in 12 months or a year? The effect of ease of estimation and confidence on budget estimates. *Journal of Consumer Research*, 35(2):245–256, Mar. 2008. 123

1844. Unicode Consortium, The. *The Unicode Standard Version 11.0 – Core Specification*. The Unicode Consortium, June 2018. 102

1845. S. S. N. Upadhyay, K. J. Houghtonb, and C. M. Klin. Is "few" always less than expected?: The influence of story context on readers' interpretation of natural language quantifiers. *Discourse Processes*, 56(8):708–727, 2018. 149

1846. D. Šmite, R. Britto, and R. van Solingen. Calculating the extra costs and the bottom-line hourly cost of offshoring. In *IEEE 12th International Conference on Global Software Engineering*, ICGSE'17, pages 96–105, May 2017. 123

1847. I. Utting, D. Bouvier, M. Caspersen, A. E. Tew, R. Frye, Y. B.-D. Kolikant, M. McCracken, J. Paterson, J. Sorva, L. Thomas, and T. Wilusz. A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the ITiCSE working group reports conference on Innovation and Technology in Computer Science Education-Working Group Reports*, ITiCSE-WGR'13, pages 15–32, June 2013. 357

1848. Ž Antolić. Fault slip through measurement process implementation in CPP software verification. In *International Conference on Business Intelligence Systems*, miproBIS 2007, May 2007. 164

1849. A. Vahabzadeh, A. M. Fard, and A. Mesbah. An empirical study of bugs in test code. In *International Conference on Software Maintenance and Evolution*, ICSME 2015, pages 101–110, Oct. 2015. 168

1850. M. Välimäki. Dual licensing in open source software industry. *Systemes d'Information et Management*, 8(1):63–75, 2003. 64

1851. J. van Angeren, C. Alves, and S. Jansen. Can we ask you to collaborate? Analyzing app developer relationships in commercial platform ecosystems. *Journal of Systems and Software*, 113:430–445, Mar. 2016. 87, 88

1852. O. Van den Bergh, S. Vrana, and P. Eelen. Letters from the heart: Affective categorization of letter combinations in typists and nontypists. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 16(6):1153–1161, 1990. 190

1853. K. G. van den Boogaart and R. Tolosana-Deldago. *Analyzing Compositional Data with R*. Springer, 2013. 344

1854. J.-B. van der Henst, L. Carles, and D. Sperber. Truthfulness and relevance in telling the time. *Mind & Language*, 17(5):457–466, Nov. 2002. 47

1855. E. van der Kouwe, D. Andriesse, H. Bos, C. Giuffrida, and G. Heiser. Benchmarking crimes: An emerging threat in systems security. In *eprint arXiv:cs.CR/1801.02381*, Jan. 2018. 362

1856. C. van der Merwe. An engineering approach to an integrated value proposition design framework. Thesis (m.s.), Faculty of Industrial Engineering at Stellenbosch University, Mar. 2015. 83

1857. M. J. P. van der Meulen. *The Effectiveness of Software Diversity*. PhD thesis, Centre for Software Reliability, City University, Nov. 2007. 126, 174, 192, 239

1858. M. J. P. van der Meulen, P. G. Bishop, and M. Revilla. An exploration of software faults and failure behaviour in a large population of programs. In *15th International Symposium on Software Reliability Engineering*, ISSRE 2004, pages 101–120, Nov. 2004. 158, 159

1859. M. P. van Oeffelen and P. G. Vos. A probabilistic model for the discrimination of visual number. *Perception & Psychophysics*, 32(2):163–170, 1982. 46

1860. K. E. van Oorschot, J. W. M. Bertrand, and C. G. Rutte. Field studies into the dynamics of product development tasks. *International Journal of Operations & Production Management*, 25(8):720–739, 2005. 130, 131

1861. P. Van Roy. Programming paradigms for dummies: What every programmer should know. In G. Assayag and A. Gerzso, editors, *New computational paradigms for computer music*, chapter 2. Delatour France, Jan. 2009. 191

1862. H. VanLehn. *Mind Bugs: The Origins of Procedural Misconceptions*. The MIT Press, 1990. 21, 46

1863. Y. Vardi and E. Weitz. *Misbehavior in Organizations: Theory, Research, and Management*. Lawrence Erlbaum Associates, Sept. 2004. 74

1864. R. Vasa. *Growth and Change Dynamics in Open Source Software Systems*. PhD thesis, Faculty of Information and Communication Technology, Swinburne University of Technology, Melbourne, Oct. 2010. 107, 254, 287

1865. B. Vasilescu, A. Serebrenik, M. Goeminne, and T. Mens. On the variation and specialisation of workload-A case study of the Gnome ecosystem community. *Empirical Software Engineering*, 19(4):955–1008, Aug. 2012. 296

1866. C. Vassallo, G. Grano, F. Palomba, H. C. Gall, and A. Bacchelli. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming*, 180:1–15, July 2019. 135

1867. P. Vassiliadis, M.-R. Kolozoff, M. Zerva, and A. V. Zarras. Schema evolution and foreign keys: a study on usage, heartbeat of change and relationship of foreign keys to table activity. *Computing*, 101(10):1431–1456, Oct. 2019. 141

1868. VCDB. VERIS community database. https://github.com/vz-risk/VCDB, Mar. 2018. 147

1869. W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, fourth edition, 2002. 379

1870. C. Vendome, M. Linares-Vásquez, G. Bavota, M. Di Penta, D. German, and D. Poshyvanyk. License usage and changes: A large-scale study on GitHub. *Empirical Software Engineering*, 22(3):1537–1577, June 2017. 65

1871. P. Verghese and D. G. Pelli. The information capacity of visual attention. *Vision Research*, 32(5):983–995, 1992. 55

1872. C. Verhoef. Quantitative IT portfolio management. *Science of Computer Programming*, 45(1):1–96, Oct. 2002. 124

1873. C. Vesel. Language bias in accident investigation. Thesis (m.s.), Lund University, Sweden, May 2012. 157

1874. I. Vessey. Cognitive fit: A theory-based analysis of the graphs versus tables literature. *Decision Sciences*, 22(2):219–240, Mar. 1991. 220

1875. A. Vetroò, R. Dürre, M. Conoscenti, D. M. Fernández, and M. Jørgensen. Combining data analytics with team feedback to improve the estimation process in agile software development. *Foundations of Computing and Decision Sciences*, 43(4):305–334, Dec. 2018. 134

1876. B. Veytsman and L. Akhmadeeva. Towards evidence-based typography: First results. *TUGboat*, 33(2):156–156, Apr. 2012. 236, 237

1877. Vgchartz global yearly chart: 2005-2016. website, Feb. 2017. http://www.vgchartz.com/yearly/2016/Global. 83

1878. V. B. Viard. Information goods upgrades: Theory and evidence. *The B. E. Journal of Theoretical Economics*, 7(1):1–34, 2007. 81

1879. C. Vickrey and A. Neuringer. Pigeon reaction time, Hick's law, and intelligence. *Psychonomic Bulletin & Review*, 7(2):284–291, June 2000. 56

1880. N. M. Victor and J. H. Ausubel. DRAMs as model organisms for study of technological evolution. *Technological Forecasting & Social Change*, 69(3):243–262, Apr. 2002. 88

1881. S. Vidal, A. Bergel, J. A. Díaz-Pace, and C. Marcosa. Over-exposed classes in Java: An empirical study. *Computer Languages, Systems & Structures*, 46:1–19, Nov. 2016. 204

1882. F. Viénot, H. Brettel, and J. D. Mollon. Digital video colourmaps for checking the legibility of displays by dichromats. *COLOR research and application*, 24(4):243–252, Aug. 1999. 224

1883. V. Villard. Android version distribution history. http://www.bidouille.org/misc/androidcharts, 2015. 95, 225

1884. T. H. Vines, A. Y. K. Albert, R. L. Andrew, F. Débarre, D. G. Bock, M. T. Franklin, K. J. Gilbert, J.-S. Moore, S. Renaut, and D. J. Rennison. The availability of research data declines rapidly with article age. In *eprint arXiv:abs/1312.5670*, Dec. 2013. 9

1885. W. K. Viscusi and J. E. Aldy. The value of a statistical life: A critical review of market estimates throughout the world. Working Paper No. 9487, National Bureau of Economic Research, USA, Feb. 2003. 148

1886. R. Viseur and G. Robles. First results about motivation and impact of license changes in open source projects. In *11th IFIP WG 2.13 International Conference*, OSS 2015, pages 137–145, May 2015. 65

1887. H. M. Vollmer, J. J. McAuliffe, R. I. Hirshberg, and K. D. Moll. Organizational design – An exploratory study. R&D Studies Series AFOSR-67-2450, Stanford Research Institute, Dec. 1967. 67

1888. K. G. Volz and G. Gigerenzer. Cognitive processes in decisions under risk are not the same as in decisions under uncertainty. *frontiers in Neuroscience*, 6:105, July 2012. 51

1889. K. von Fintel and L. Matthewson. Universals in semantics. *The Linguistic Review*, 25(1-2):139–201, 2008. 40

1890. A. von Rhein, J. Liebig, A. Janker, C. Kästner, and S. Apel. Variability-aware static analysis at scale: An empirical study. *ACM Transactions on Software Engineering and Methodology*, 27(4):18, Nov. 2018. 168

1891. S. L. R. Vrhovec, T. Hovelja, D. Vavpotič, and M. Krisper. Diagnosing organizational risks in software projects: Stakeholder resistance. *International Journal of Project Management*, 33(6):1262–1273, Aug. 2015. 131

1892. M. Wachs. When planners lie with numbers. *Journal of the American Planning Association*, 55(4):476–479, Apr. 1989. 123

1893. J. Wagemans, J. H. Elder, M. Kubovy, M. A. Peterson, S. E. Palmer, M. Singh, and R. von der Heydt. A century of Gestalt psychology in visual perception: I. Perceptual grouping and figure-ground organization. *Psychological Bulletin*, 138(6):1172–1217, 2012. 25

1894. J. Wai, M. Cacchio, M. Putallaz, and M. C. Makel. Sex differences in the right tail of cognitive abilities: A 30 year examination. *Intelligence*, 38(4):412–423, July-Aug. 2010. 19

1895. J. Wainer, C. G. N. Barsottini, D. Lacerda, and L. R. M. de Marco. Empirical evaluation in computer science research published by ACM. *Information and Software Technology*, 51(6):1081–1085, June 2009. 6

1896. L. Wakeham. Government policy on the management of risk, volume I: Report. HL Paper 183-I, Select Committee on Economic Affairs, UK House of Lords, June 2006. 149

1897. S. Waligora, J. Bailey, and M. Stark. Impact of Ada and object-oriented design in the flight dynamics division at Goddard space flight center. Technical Report SEL-95-001, Goddard Space Flight Center, Mar. 1995. 192, 212

1898. D. R. Wallace and D. R. Kuhn. Failure modes in medical device software: An analysis of 15 years of recall data. *International Journal of Reliability, Quality and Safety Engineering*, 8(4):351–372, Dec. 2001. 147

1899. H. Wang, H. Li, L. Li, Y. Guo, and G. Xu. Why are Android apps removed from Google play? A large-scale empirical study. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR'18, pages 231–242, May 2018. 106

1900. P. Wang. Chasing the hottest IT: Effects of information technology fashion on organizations. *MIS Quarterly*, 34(1):63–85, Mar. 2010. 4, 9

1901. P. Wang and K. T. Stolee. How well are regular expressions tested in the wild? In *Proceedings of the 26th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, ESEC/FSE'18, pages 668–678, Nov. 2018. 168

1902. W. Wang. Toward improved understanding and management of software clones. Thesis (m.s.), University of Waterloo, Ontario, Canada, May 2012. 78

1903. Y. Wang. Language matters. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM'15, pages 1–10, Oct. 2015. 103

1904. Y. Wang and J. Zhang. The effort distribution of software development phases. *Computer Science and Application*, 7(5):428–437, May 2017. 123, 126

1905. L. Wanner, C. Apte, R. Balani, P. Gupta, and M. Srivastava. A case for opportunistic embedded sensing in presence of hardware power variability. In *Proceedings of the 2010 international conference on Power aware computing and systems*, HotPower'10, pages 1–8, Oct. 2010. 365

1906. G. Ward, L. Tan, and R. Grenfell-Essam. Examining the relationship between free recall and immediate serial recall: the effects of list length and output order. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, 36(5):1207–1241, Sept. 2010. 32

1907. P. J. Ward. Euclid's Elements, from Hilbert's axioms. Thesis (m.s.), The Ohio State University, 2012. 144

1908. C. Ware. *Information Visualization Perception for Design*. Morgan Kaufmann Publishers, 2000. 24

1909. W. H. Ware, S. N. Alexander, P. Armer, M. M. Astrahan, L. Bers, H. H. Goode, H. D. Huskey, and M. Rubinoff. Soviet computer technology–1959. Research Memorandum RM-2541, The RAND Corporation, Mar. 1960. 5

1910. P. C. Wason. On the failure to eliminate hypotheses in a conceptual task. *The Quarterly Journal of Experimental Psychology*, XII(3):129–140, 1960. 23

1911. P. C. Wason. Reasoning about a rule. *The Quarterly Journal of Experimental Psychology*, 20(3):273–281, 1968. 42

1912. J. Waters. Variable marginal propensities to pirate and the diffusion of computer software. MPRA Paper No. 46036, Nottingham University Business School, Apr. 2013. 83

1913. C. Watson and F. W. B. Li. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation technology in computer science education*, ITiCSE'14, pages 39–44, June 2014. 357

1914. V. M. Weaver and J. Dongarra. Can hardware performance counters produce expected, deterministic results? In *3rd Workshop on Functionality of Hardware Performance Monitoring*, pages 1–11, Dec. 2010. 366

1915. V. M. Weaver and S. A. McKee. Can hardware performance counters be trusted? In *IEEE International Symposium on Workload Characterization*, IISWC'08, pages 141–150, Sept. 2008. 366

1916. M. Webb, N. Bloom, N. Short, and J. Lerner. Some facts of high-tech patenting. Working Paper No. 18-023, Stanford Institute for Economic Policy Research, July 2018. 64

1917. E. U. Weber, A.-R. Blais, and N. E. Betz. A domain-specific risk-attitude scale: Measuring risk perceptions and risk behaviors. *Journal of Behavior and Decision Making*, 15(4):263–290, Apr. 2002. 51

1918. B. F. Webster. Patterns in IT litigation: Systems failure (1976-2000). A study, PriceaterhouseCoopers LLP, 2000. 120

1919. B. S. Weekes. Differential effects of number of letters on word and nonword naming latency. *The Quarterly Journal of Experimental Psychology*, 50A(2):439–456, 1997. 30

1920. D. M. Wegner. *The Illusion of Conscious Will*. MIT Press, 2002. 18

1921. M. H. Weik. A survey of domestic electronic digital computing systems. Technical Report 971, Ballistic Research Laboratories, Maryland, Dec. 1955. 108, 361

1922. M. H. Weik. A third survey of domestic electronic digital computing systems. Technical Report 1115, Ballistic Research Laboratories, Maryland, Mar. 1961. 108, 361

1923. G. F. Weinwurm and H. J. Zagorski. Research into the management of computer programming: A transitional analysis of cost estimation techniques. Technical Documentary Report ESD-TR-65-575, United States Air Force, L. G. Hanscom Field, Bedford, Massachusetts, Nov. 1965. 123

1924. M. V. Welser. Opposing the monetization of Linux: McHardy v. Geniatech & addressing copyright "trolling" in Germany. *International Free and Open Source Software Law Review*, 10(1):9–20, 2018. 66

1925. J. West and J. Dedrick. Innovation and control in standards architectures: The rise and fall of Japan's PC-98. *Information Systems Research*, 11(2):197–216, June 2000. 92

1926. J. A. White. Grapher pics. `http://www.talljerome.com/mathnerd.html`, Oct. 2012. 227

1927. M. White. Scaled CMOS technology reliability users guide. JPL Publication 09-33 01/10, Jet Propulsion Laboratory, California Institute of Technology, 2010. 4, 161

1928. White House, The. Guidelines and discount rates for benefit-cost analysis of federal programs. OMB Circular A-94, U.S. Government, 1992. 60

1929. D. Whitfield. Cost overruns, delays and terminations: 105 outsourced public sector ICT projects. ESSU Research Report 3, European Services Strategy Unit, Dec. 2007. 118

1930. R. M. Whyte. Order Re Sun's Motions for Preliminary Injunction Against Microsoft. Re: Sun Microsystems v. Microsoft, Case No. 97-20884 RMW(PVT). Opinion, UNITED STATES DISTRICT COURT FOR THE NORTHERN DISTRICT OF CALIFORNIA, 1998. 106, 167

1931. W. F. Whyte. *Money and Motivation: An Analysis of Incentives in Industry*. Harper Torchbooks, Jan. 1970. 69, 70

1932. J. M. Wicherts, M. Bakker, and D. Molenaar. Willingness to share research data is related to the strength of the evidence and the quality of reporting of statistical results. *PLoS ONE*, 6(11):e26828, Nov. 2011. 9

1933. W. A. Wickelgren. Size of rehearsal group and short-term memory. *Journal of Experimental Psychology*, 68(4):413–419, 1964. 31

1934. G. Wiederhold. What is your software worth? Technical Report ???, Stanford University, Apr. 2007. 79

1935. A. Wierzbicka. *Semantics: Primes and Universals*. Oxford University Press, 1996. 40

1936. I. S. Wiese, J. T. da Silva, I. Steinmacher, C. Treude, and M. A. Gerosa. Who is who in the mailing list? Comparing six disambiguation heuristics to identify multiple addresses of a participant. In *International Conference on Software Maintenance and Evolution*, ICSME 2016, pages 345–355, Oct. 2016. 380

1937. Wikipedia. List of most expensive video games to develop. website, 2018. `https://en.wikipedia.org/List_of_most_expensive_video_games_to_develop`. 59

1938. R. Wilcox. *Introduction to Robust Estimation & Hypothesis Testing*. Elsevier, 3rd edition, 2012. 259

1939. J. Wiley. Expertise as mental set: The effects of domain knowledge in creative problem solving. *Memory & Cognition*, 26(4):716–730, 1998. 38

1940. M. V. Wilkes. *Memoirs of a Computer Pioneer*. MIT Press, 1984. 143

1941. M. V. Wilkes, D. J. Wheeler, and S. Gill. *The Preparation of Programs for an Electronic Digital Computer*. Addison–Wesley, second edition, 1957. 111

1942. J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Dover Publications, 1994. 146

1943. L. Wilkinson. *The Grammar of Graphics*. Springer, second edition, 2005. 221

1944. J. C. Williams. A data-based method for assessing and reducing human error to improve operational performance. In *Fourth Conference on Human Factors and Power Plants*, pages 436–450, June 1988. 21

1945. P. Williams and B. Curtis. A matched project evaluation of modern programming practices: Scientific report on the ASTROS plan. Technical Report RADC-TR-80-6, Vol II, General Electric Company, Feb. 1980. 137

1946. R. R. Willis, R. M. Rova, M. D. Scott, M. I. Johnson, J. F. Ryskowski, J. A. Moon, K. C. Shumate, and T. O. Winfield. Hughes Aircraft's widespread deployment of a continuously improving software process. Technical Report CMU/SEI-98-TR-006, Raytheon Systems Company, May 1998. 79

1947. H. E. Willman, Jr., T. A. James, A. A. Beaureguard, and P. Hilcoff. Software systems reliability: A Raytheon project history. Final Technical Report RADC-TR-77-188, Rome Air Development Center, Griffiss Air Force Base, June 1977. 147

1948. L. M. Wills. Automated program recognition by graph parsing. A.I. Technical Report No. 1358, MIT Artificial Intelligence Laboratory, July 1992. 180

1949. M. P. Wilmot and D. S. Ones. A century of research on conscientiousness at work. *PNAS*, 116(46):23004–23010, Nov. 2019. 54

1950. R. Wiltbank and W. Boeker. Returns to angel investors in groups. Working Paper 1028592, US universities, Nov. 2007. 89

1951. K. Winter, H. Femmer, and A. Vogelsang. How do quantifiers affect the quality of requirements? In *eprint arXiv:cs.SE/2002.02672*, Feb. 2014. 158, 159

1952. J. C. Wise, D. L. Hannaman, P. Kozumplik, E. Franke, and B. L. Leaver. Methods to improve cultural communication skills in special operations forces. ARI Contract Report 98-06, United States Army Research Institute for the Behavioral and Social Sciences, July 1998. 102

1953. K. Wnuk, J. Kabbedijk, S. Brinkkemper, B. Regnell, and D. Callele. Factors affecting decision outcome and lead-time in large-scale requirements engineering. *Journal of Software: Evolution and Process*, 27(9):647–673, Sept. 2015. 129

1954. C. Wohlin, P. Runeson, and J. Brantestam. An experimental evaluation of capture-recapture in software inspections. *Journal of Software Testing, Verification and Reliability*, 5(4):213–232, 1995. 372

1955. R. W. Wolverton. The cost of developing large-scale software. *IEEE Transactions on Computers*, c-23(6):615–636, June 1974. 127

1956. W. E. Wong, S. S. Gokhale, and J. R. Horgan. Quantifying the closeness between program components and features. *The Journal of Systems and Software*, 54(2):87–98, Oct. 2000. 180

1957. A. Wood. Software reliability growth models. Technical Report 96.1, Tandem Computer, Sept. 1996. 153, 155

1958. R. Woodfield. Undergraduate retention and attainment across the disciplines. Report, The Higher Education Academy, York, UK, Dec. 2014. 357

1959. World Semiconductor Trade Statistics. Semiconductor monthly sales volume: 1975–2016. website, Mar. 2016. `https://www.wsts.org`. 5

1960. D. Wren. Passmark website. `http://www.passmark.com`, July 2014. 370, 372

1961. J. D. Wren, A. Valencia, and J. Kelso. Reviewer-coerced citation: case report, update on journal policy and suggestions for future prevention. *Bioinformatics*, 35(18):3217–3218, Sept. 2019. 9

1962. R. Wright. *The Evolution of GOD*. Little, Brown Book Group, 2009. 92

1963. A. Wrzesniewski, C. McCauley, P. Rozin, and B. Schwartz. Jobs, careers, and callings: People's relations to their work. *Journal of Research in Personality*, 31(1):21–33, Mar. 1997. 70

1964. S. D. Wu, C. Rossin, K. G. Kempf, M. O. Atan, B. Aytac, S. A. Shirodkar, and A. Mishra. Extending Bass for improved new product forecasting. Wagner Prize, Apr. 2009. 82, 83

1965. G. Xiao, Z. Zheng, B. Jiang, and Y. Sui. An empirical study of regression bug chains in Linux. *IEEE Transactions on Reliability*, 69(2):558–570, June 2020. 147

1966. J. Yan and W. Zhang. Compiler-guided register reliability improvement against soft errors. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT'05, pages 203–209, Sept. 2005. 163

1967. M. Yang, G.-R. Uh, and D. B. Whalley. Efficient and effective branch reordering using profile data. *ACM Transactions on Programming Languages and Systems*, 24(6):667–697, Nov. 2002. 199

1968. M. C. K. Yang and A. Chao. Reliability-estimation & stopping-rules for software testing, based on repeated appearances of bugs. *IEEE Transactions on Reliability*, 44(2):315–321, June 1995. 171

1969. X. Yang, Z. Wang, J. Xue, and Y. Zhou. The reliability wall for exascale supercomputing. *IEEE Transactions on Computers*, 61(6):767–779, June 2011. 163

1970. Y. Yang, Y. Zhou, H. Sun, Z. Su, Z. Zuo, L. Xu, and B. Xu. Hunting for bugs in code coverage tools via randomized differential testing. In *IEEE/ACM 41st International Conference on Software Engineering*, ICSE'19, pages 488–499, May 2019. 161, 354

1971. M. J. Yap, S. J. R. Liow, S. B. Jalil, and S. S. B. Faizal. The Malay lexicon project: A database of lexical statistics for 9,592 words. *Behavior Research Methods*, 42(4):992–1003, Nov. 2010. 191

1972. J. Yates. *Structuring the Information Age: Life Insurance and technology in the Twentieth Century*. The Johns Hopkins University Press, Nov. 2008. 101

1973. Y. C. B. Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings Aerospace Applications Conference (vol 1)*, pages 293–307, Feb. 1996. 162

1974. J. R. Yost. *Making IT Work: A History of the Computer Services Industry*. The MIT Press, 2017. 101

1975. A. G. Yu. *Managing Application Software Suppliers in Information System Development Projects*. PhD thesis, Department of Management and Organisation, University of Stirling, Nov. 2003. 127, 128

1976. L. Yu and S. Ramaswamy. A study of SourceForge users and user network. AAAI Technical Report FS-13-05, Association for the Advancement of Artificial Intelligence, Nov. 2013. 198

1977. D. Yuan, S. Park, and Y. Zhou. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE'12, pages 102–112, June 2012. 163

1978. T. Yuki and S. Rajopadhye. Folklore confirmed: Compiling for speed = compiling for energy. Technical Report CS13-107, Computer Science Department, Colorado State University, Aug. 2013. 364

1979. A. Zaidman, B. V. Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Technical Report TUD-SERG-2010-035, Software Engineering Research Group, Delft University of Technology, 2010. 168

1980. M. J. Zbaracki. The rhetoric and reality of total quality management. *Administrative Science Quarterly*, 43(3):602–636, Sept. 1998. 74

1981. S. F. Zeigler. Comparing development costs of C and Ada. Technical report, Rational Software Corporation, Mar. 1995. 55

1982. A. Zeileis, K. Hornik, and P. Murrell. Escaping RGBland: Selecting colors for statistical graphics. *Computational Statistics & Data Analysis*, 53(9):3259–3270, July 2009. 224

1983. M. V. Zelkowitz. The effectiveness of software prototyping: A case study. In *ACM Washington Chapter 26th Annual Technical Symposium*, pages 7–15, June 1987. 129

1984. A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. *The Fuzzing Book: Tools and Techniques for Generating Software Tests*. ???, Dec. 2019. 167

1985. A. Zeller, T. Zimmermann, and C. Bird. Failure is a four-letter word-A parody in empirical research-. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, PROMISE'11, pages 5:1–5:7, Sept. 2011. 264, 278

1986. A. Zerouali and T. Mens. Analyzing the evolution of testing library usage in open source Java projects. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, SANER 2017, pages 503–507, Feb. 2017. 141

1987. J. Zhang and H. Wang. The effect of external representations on numeric tasks. *The Quarterly Journal of Experimental Psychology*, 58(5):817–838, Oct. 2005. 47

1988. J. Zhang, M. Zhu, D. Hao, and L. Zhang. An empirical study on the scalability of selective mutation testing. In *Proceedings 25th International Symposium on Software Reliability Engineering*, ISSRE'14, pages 277–287, Nov. 2014. 171

1989. Q. Zhang, C. Sun, and Z. Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'17, pages 347–361, June 2017. 167

1990. T. Zhang, D. Yang, C. Lopes, and M. Kim. Analyzing and supporting adaptation of online code examples. In *eprint arXiv:cs.SE/1905.12111*, May 2019. 64, 65

1991. X. Zhang. *An Analysis of the Effect of Environmental and Systems Complexity on Information Systems Failures*. PhD thesis, University of North Texas, Aug. 2001. 98

1992. Y. Zhang, Y. Jiang, C. Xu, X. Ma, and P. Yu. ABC: Accelerated building of C/C++ projects. In *Asia-Pacific Software Engineering Conference*, APSEC 2015, pages 182–189, Dec. 2015. 194

1993. Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August. DAFT: Decoupled acyclic fault tolerance. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT'10, pages 87–98, Sept. 2010. 162

1994. M. Zhao, J. Grossklags, and P. Liu. An empirical study of web vulnerability discovery ecosystems. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS'15, pages 1105–1117, Oct. 2015. 105, 147

1995. M. Zhao and P. Liu. Empirical analysis and modeling of black-box mutational fuzzing. In *International Symposium on Engineering Secure Software and Systems*, ESSoS 2016, pages 173–189, Apr. 2016. 154, 156

1996. Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu. The impact of continuous integration on other software development practices: A large-scale empirical study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE'17, pages 60–71, Oct.-Nov. 2017. 136

1997. J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, Apr. 2006. 166

1998. Q. Zheng, A. Mockus, and M. Zhou. A method to identify and correct problematic software activity data: Exploiting capacity constraints and data redundancies. In *Proceedings of the 10th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, ESEC/FSE'15, pages 637–648, Aug.-Sept. 2015. 376

1999. H. Zhong and Z. Su. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE'15, pages 913–923, May 2015. 160, 161

2000. H. Zhou and A. Fishbach. The pitfall of experimenting on the web: How unattended selective attrition leads to surprising (yet false) research conclusions. *Journal of Personality and Social Psychologs*, 111(4):493–504, Oct. 2016. 357

2001. J. Zhou, S. Wang, C.-P. Bezemer, Y. Zou, and A. E. Hassan. Bounties in open source development on GitHub: A case study of Bountysource bounties. In *eprint arXiv:cs.SE/1904.02724*, Apr. 2019. 150, 151

2002. K. Zhou, P. Huang, C. Li, and H. Wang. An empirical study on the interplay between filesystems and SSD. In *7th International Conference on Networking, Architecture and Storage*, NAS'12, pages 124–133, June 2012. 369, 370

2003. S. Zhou, B. Vasilescu, and C. Kästner. What the fork: A study of inefficient and efficient forking practices in social coding. In *Proceedings of the 27th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, ESEC/FSE'19, pages 350–361, Aug. 2019. 141

2004. Y. Zhou, L. Wu, Z. Wang, and X. Jiang. Harvesting developer credentials in Android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec'15, page 23, June 2015. 202

2005. X. Zhu, E. J. Whitehead, Jr., C. Sadowski, and Q. Song. An analysis of programming language statement frequency in C, C++, and Java source code. *Software–Practice and Experience*, 15(11):1479–1495, Nov. 2015. 237, 238

2006. A. Ziegler, V. Rothberg, and D. Lohmann. Analyzing the impact of feature changes in Linux. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS'16, pages 25–32, Jan. 2016. 193

2007. T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, ESEC/FSE 2009, pages 91–100, Aug. 2009. 165

2008. T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE'07, May 2007. 160

2009. J. O. Zinn. The proliferation of 'at risk' in The Times: A corpus approach to historical social change, 1785-2009. *Historical Social Research*, 43(2):313–364, 2018. 148

2010. P. M. Zislis. An experiment in algorithm implementation. Technical Report CSD-TR 96, Purdue University, June 1973. 35, 36, 192, 194

2011. F. Zlotnick. *The POSIX.1 Standard: A Programmer's Guide*. The Benjamin/Cummings Publishing Company, 1991. 111

2012. W. Zou, W. Zhang, X. Xia, R. Holmes, and Z. Chen. Branch use in practice: A large-scale empirical study of 2,923 projects on GitHub. In *19th International Conference on Software Quality, Reliability and Security*, QRS 2019, pages 306–317, July 2019. 134, 135

2013. K. Zuse. Über den allgemeinen Plankalkül als mittel zur formulierung schematisch-kombinativer aufgaben. *Archiv der Mathematik*, 1:441–449, 1949. 109

2014. O. Zwikael and S. Globerson. Benchmarking of project planning and success in selected industries. *Benchmarking: An International Journal*, 13(6):688–700, 2006. 116