Chapter 3

Data Munging

> On two occasions I have been asked, "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" ... I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.
>
> – Charles Babbage

Most data scientists spend much of their time cleaning and formatting data. The rest spend most of their time complaining that there is no data available to do what they want to do.

In this chapter, we will work through some of the basic mechanics of computing with data. Not the high-faluting stuff like statistics or machine learning, but the grunt work of finding data and cleaning it that goes under the moniker of *data munging*.

While practical questions like "What is the best library or programming language available?" are clearly important, the answers change so rapidly that a book like this one is the wrong place to address them. So I will stick at the level of general principles, instead of shaping this book around a particular set of software tools. Still, we will discuss the landscape of available resources in this chapter: why they exist, what they do, and how best to use them.

The first step in any data science project is getting your hands on the right data. But this is often distressingly hard. This chapter will survey the richest hunting grounds for data resources, and then introduce techniques for cleaning what you kill. Wrangling your data so you that can safely analyze it is critical for meaningful results. As Babbage himself might have said more concisely, "garbage in, garbage out."

3.1 Languages for Data Science

In theory, every sufficiently powerful programming language is capable of expressing any algorithm worth computing. But in practice, certain programming

© The Author(s) 2017
S.S. Skiena, *The Data Science Design Manual,*
Texts in Computer Science, https://doi.org/10.1007/978-3-319-55444-0_3

languages prove much better than others at specific tasks. Better here might denote *easier for the programmer* or perhaps *more computationally efficient*, depending upon the mission at hand.

The primary data science programming languages to be aware of are:

- *Python*: This is today's bread-and-butter programming language for data science. Python contains a variety of language features to make basic data munging easier, like regular expressions. It is an interpreted language, making the development process quicker and enjoyable. Python is supported by an enormous variety of libraries, doing everything from scraping to visualization to linear algebra and machine learning.

  Perhaps the biggest strike against Python is efficiency: interpreted languages cannot compete with compiled ones for speed. But Python compilers exist in a fashion, and support linking in efficient C/assembly language libraries for computationally-intensive tasks. Bottom line, Python should probably be your primary tool in working through the material we present in this book.

- *Perl*: This *used* to be the go to language for data munging on the web, before Python ate it for lunch. In the TIOBE programming language popularity index (`http://www.tiobe.com/tiobe-index`), Python first exceeded Perl in popularity in 2008 and hasn't looked back. There are several reasons for this, including stronger support for object-oriented programming and better available libraries, but the bottom line is that there are few good reasons to start projects in Perl at this point. Don't be surprised if you encounter it in some legacy project, however.

- *R*: This is the programming language of statisticians, with the deepest libraries available for data analysis and visualization. The data science world is split between R and Python camps, with R perhaps more suitable for exploration and Python better for production use. The style of interaction with R is somewhat of an acquired taste, so I encourage you to play with it a bit to see whether it feels natural to you.

  Linkages exist between R and Python, so you can conveniently call R library functions in Python code. This provides access to advanced statistical methods, which may not be supported by the native Python libraries.

- *Matlab*: The Mat here stands for *matrix*, as Matlab is a language designed for the fast and efficient manipulation of matrices. As we will see, many machine learning algorithms reduce to operations on matrices, making Matlab a natural choice for engineers programming at a high-level of abstraction.

  Matlab is a proprietary system. However, much of its functionality is available in GNU Octave, an open-source alternative.

- *Java and C/C++*: These mainstream programming languages for the development of large systems are important in big data applications. Parallel processing systems like Hadoop and Spark are based on Java and C++, respectively. If you are living in the world of distributed computing, then you are living in a world of Java and C++ instead of the other languages listed here.

- *Mathematica/Wolfram Alpha*: Mathematica is a proprietary system providing computational support for all aspects of numerical and symbolic mathematics, built upon the less proprietary Wolfram programming language. It is the foundation of the Wolfram Alpha computational knowledge engine, which processes natural language-like queries through a mix of algorithms and pre-digested data sources. Check it out at `http://www.wolframalpha.com`.

  I will confess a warm spot for Mathematica.[1] It is what I tend to reach for when I am doing a small data analysis or simulation, but cost has traditionally put it out of the range of many users. The release of the Wolfram language perhaps now opens it up to a wider community.

- *Excel*: Spreadsheet programs like Excel are powerful tools for exploratory data analysis, such as playing with a given data set to see what it contains. They deserve our respect for such applications.

  Full featured spreadsheet programs contain a surprising amount of hidden functionality for power users. A student of mine who rose to become a Microsoft executive told me that 25% of all new feature requests for Excel proposed functionality already present there. The special functions and data manipulation features you want probably are in Excel if you look hard enough, in the same way that a Python library for what you need probably will be found if you search for it.

3.1.1 The Importance of Notebook Environments

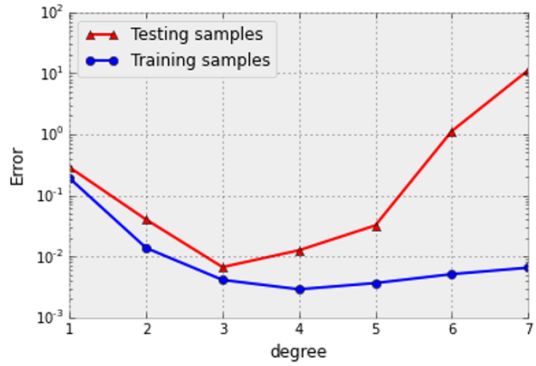
The primary deliverable for a data science project should not be a program. It should not be a data set. It should not be the results of running the program on your data. It should not just be a written report.

The deliverable result of every data science project should be a computable notebook tying together the code, data, computational results, and written analysis of what you have learned in the process. Figure 3.1 presents an excerpt from a Jupyter/IPython notebook, showing how it integrates code, graphics, and documentation into a descriptive document which can be executed like a program.

The reason this is so important is that computational results are the product of long chains of parameter selections and design decisions. This creates several problems that are solved by notebook computing environments:

[1] Full disclosure: I have known Stephen Wolfram for over thirty years. Indeed, we invented the iPad together [Bar10, MOR+88].

In [40]:
```
degrees = range(1, 8)
errors = np.array([regressor3(d) for d in degrees])
plt.plot(degrees, errors[:, 0], marker='^', c='r', label='Testing samples
plt.plot(degrees, errors[:, 1], marker='o', c='b', label='Training sample
plt.yscale('log')
plt.xlabel("degree"); plt.ylabel("Error")
  = plt.legend(loc='best')
```



By sweeping the degree we discover two regions of model performance:

- **Underfitting** (degree < 3): Characterized by the fact that the testing error will get lower if we increase the model capacity.
- **Overfitting** (degree > 3): Characterized by the fact the testing will get higher if we increase the model capacity. Note, that the training error is getting lower or just staying the same!.

Figure 3.1: Jupyter/IPython notebooks tie together code, computational results, and documentation.

- Computations need to be *reproducible*. We must be able to run the same programs again from scratch, and get exactly the same result. This means that data pipelines must be *complete*: taking raw input and producing the final output. It is terrible karma to start with a raw data set, do some processing, edit/format the data files by hand, and then do some *more* processing – because what you did by hand cannot be readily done again on another data set, or undone after you realize that you may have goofed up.
- Computations must be *tweakable*. Often reconsideration or evaluation will prompt a change to one or more parameters or algorithms. This requires rerunning the notebook to produce the new computation. There is nothing more disheartening to be given a big data product without provenance and told that *this* is the final result and you can't change anything. A notebook is never finished until after the entire project is done.
- Data pipelines need to be *documented*. That notebooks permit you to integrate text and visualizations with your code provides a powerful way to communicate what you are doing and why, in ways that traditional programming environments cannot match.

Take-Home Lesson: Use a notebook environment like IPython or Mathematica to build and report the results of any data science project.

3.1.2 Standard Data Formats

Data comes from all sorts of places, and in all kinds of formats. Which representation is best depends upon who the ultimate consumer is. Charts and graphs are marvelous ways to convey the meaning of numerical data to people. Indeed, Chapter 6 will focus on techniques for visualizing data. But these pictures are essentially useless as a source of data to compute with. There is a long way from printed maps to Google Maps.

The best computational data formats have several useful properties:

- *They are easy for computers to parse*: Data written in a useful format is destined to be used again, elsewhere. Sophisticated data formats are often supported by APIs that govern technical details ensuring proper format.
- *They are easy for people to read*: Eyeballing data is an essential operation in many contexts. Which of the data files in this directory is the right one for me to use? What do we know about the data fields in this file? What is the gross range of values for each particular field?

  These use cases speak to the enormous value of being able to open a data file in a text editor to look at it. Typically, this means presenting the data in a human-readable text-encoded format, with records demarcated by separate lines, and fields separated by delimiting symbols.

- *They are widely used by other tools and systems*: The urge to invent proprietary data standard beats firmly in the corporate heart, and most software developers would rather share a toothbrush than a file format. But these are impulses to be avoided. The power of data comes from mixing and matching it with other data resources, which is best facilitated by using popular standard formats.

One property I have omitted from this list is *conciseness*, since it is generally not a primary concern for most applications running on modern computing systems. The quest to minimize data storage costs often works against other goals. Cleverly packing multiple fields into the higher-order bits of integers saves space, but at the cost of making it incompatible and unreadable.

General compression utilities like gzip prove amazingly good at removing the redundancy of human-friendly formatting. Disk prices are unbelievably cheap: as I write this you can buy a 4TB drive for about $100, meaning less than the cost of one hour of developer time wasted programming a tighter format. Unless you are operating at the scale of Facebook or Google, conciseness does not have nearly the importance you are liable to think it does.[2]

The most important data formats/representations to be aware of are discussed below:

- *CSV (comma separated value) files*: These files provide the simplest, most popular format to exchange data between programs. That each line represents a single record, with fields separated by commas, is obvious from inspection. But subtleties revolve around special characters and text strings: what if your data about names contains a comma, like "Thurston Howell, Jr." The csv format provides ways to escape code such characters so they are not treated as delimiters, but it is messy. A better alternative is to use a rarer delimiter character, as in tsv or *tab separated value* files.

  The best test of whether your csv file is properly formatted is whether Microsoft Excel or some other spreadsheet program can read it without hassle. Make sure the results of every project pass this test as soon as the first csv file has been written, to avoid pain later.

- *XML (eXtensible Markup Language)*: Structured but non-tabular data are often written as text with annotations. The natural output of a named-entity tagger for text wraps the relevant substrings of a text in brackets denoting person, place, or thing. I am writing this book in LaTex, a formatting language with bracketing commands positioned around mathematical expressions and *italicized text*. All webpages are written in HTML, the hypertext markup language which organizes documents using bracketing commands like `<b>` and `</b>` to enclose **bold faced text**.

  XML is a language for writing specifications of such markup languages. A proper XML specification enables the user to parse any document complying with the specification. Designing such specifications and fully adhering

[2] Indeed, my friends at Google assure me that they are often slovenly about space even at the petabyte scale.

to them requires discipline, but is worthwhile. In the first version of our Lydia text analysis system, we wrote our markups in a "pseudo-XML," read by ad hoc parsers that handled 99% of the documents correctly but broke whenever we tried to extend them. After a painful switch to XML, everything worked more reliably *and* more efficiently, because we could deploy fast, open-source XML parsers to handle all the dirty work of enforcing our specifications.

- *SQL (structured query language) databases*: Spreadsheets are naturally structured around single tables of data. In contrast, relational databases prove excellent for manipulating multiple distinct but related tables, using SQL to provide a clunky but powerful query language.

  Any reasonable database system imports and exports records as either csv or XML files, as well as an internal content dump. The internal representation in databases is opaque, so it really isn't accurate to describe them as a data format. Still, I emphasize them here because SQL databases generally prove a better and more powerful solution than manipulating multiple data files in an ad hoc manner.

- *JSON (JavaScript Object Notation)*: This is a format for transmitting data objects between programs. It is a natural way to communicate the state of variables/data structures from one system to another. This representation is basically a list of attribute-value pairs corresponding to variable/field names, and the associated values:

```
{"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]}
```

  Because library functions that support reading and writing JSON objects are readily available in all modern programming languages, it has become a very convenient way to store data structures for later use. JSON objects are human readable, but are quite cluttered-looking, representing arrays of records compared to CSV files. Use them for complex structured objects, but not simple tables of data.

- *Protocol buffers*: These are a language/platform-neutral way of serializing structured data for communications and storage across applications. They are essentially lighter weight versions of XML (where you define the format of your structured data), designed to communicate small amounts of data across programs like JSON. This data format is used for much of the inter-machine communication at Google. Apache Thrift is a related standard, used at Facebook.

3.2 Collecting Data

The most critical issue in any data science or modeling project is finding the right data set. Identifying viable data sources is an art, one that revolves around three basic questions:

- Who might actually have the data I need?
- Why might they decide to make it available to me?
- How can I get my hands on it?

In this section, we will explore the answers to these questions. We look at common sources of data, and what you are likely to be able to find and why. We then review the primary mechanisms for getting access, including APIs, scraping, and logging.

3.2.1 Hunting

Who has the data, and how can you get it? Some of the likely suspects are reviewed below.

Companies and Proprietary Data Sources

Large companies like Facebook, Google, Amazon, American Express, and Blue Cross have amazing amounts of exciting data about users and transactions, data which could be used to improve how the world works. The problem is that getting outside access is usually impossible. Companies are reluctant to share data for two good reasons:

- Business issues, and the fear of helping their competition.
- Privacy issues, and the fear of offending their customers.

A heartwarming tale of what can happen with corporate data release occurred when AOL provided academics with a data set of millions of queries to its search engine, carefully stripped of identifying information. The first thing the academics discovered was that the most frequently-entered queries were desperate attempts to escape to other search engines like Google. This did nothing to increase public confidence in the quality of AOL search.

Their second discovery was that it proved much harder to anonymize search queries than had previously been suspected. Sure you can replace user names with id numbers, but it is not that hard to figure out who the guy on Long Island repeatedly querying *Steven Skiena*, *Stony Brook*, and `https://twitter.com/search?q=Skiena&src=sprv` is. Indeed, as soon as it became publicized that people's identities had been revealed by this data release, the responsible party was fired and the data set disappeared. User privacy is important, and ethical issues around data science will be discussed in Section 12.7.

So don't think you are going to sweet talk companies into releasing confidential user data. However, many responsible companies like *The New York Times*, Twitter, Facebook, and Google do release certain data, typically by rate-limited application program interfaces (APIs). They generally have two motives:

- Providing customers and third parties with data that can increase sales. For example, releasing data about query frequency and ad pricing can encourage more people to place ads on a given platform.

- It is generally better for the company to provide well-behaved APIs than having cowboys repeatedly hammer and scrape their site.

So hunt for a public API before reading Section 3.2.2 on scraping. You won't find exactly the content or volume that you dream of, but probably something that will suffice to get started. Be aware of limits and terms of use.

Other organizations do provide bulk downloads of interesting data for offline analysis, as with the Google Ngrams, IMDb, and the taxi fare data sets discussed in Chapter 1. Large data sets often come with valuable metadata, such as book titles, image captions, and edit history, which can be re-purposed with proper imagination.

Finally, most organizations have internal data sets of relevance to their business. As an employee, you should be able to get privileged access while you work there. Be aware that companies have internal data access policies, so you will still be subject to certain restrictions. Violating the terms of these policies is an excellent way to become an ex-employee.

Government Data Sources

Collecting data is one of the important things that governments do. Indeed, the requirement that the United States conduct a census of its population is mandated by our constitution, and has been running on schedule every ten years since 1790.

City, state, and federal governments have become increasingly committed to open data, to facilitate novel applications and improve how government can fulfill its mission. The website `http://Data.gov` is an initiative by the federal government to centrally collect its data sources, and at last count points to over 100,000 data sets!

Government data differs from industrial data in that, in principle, it belongs to the People. The *Freedom of Information Act* (FOI) enables any citizen to make a formal request for any government document or data set. Such a request triggers a process to determine what can be released without compromising the national interest or violating privacy.

State governments operate under fifty different sets of laws, so data that is tightly held in one jurisdiction may be freely available in others. Major cities like New York have larger data processing operations than many states, again with restrictions that vary by location.

I recommend the following way of thinking about government records. If you cannot find what you need online after some snooping around, figure out which agency is likely to have it. Make a friendly call to them to see if they can help you find what you want. But if they stonewall you, feel free to try for a FOI act request. Preserving privacy is typically the biggest issue in deciding whether a particular government data set can be released.

Academic Data Sets

There is a vast world of academic scholarship, covering all that humanity has deemed worth knowing. An increasing fraction of academic research involves the creation of large data sets. Many journals now require making source data available to other researchers prior to publication. Expect to be able to find vast amounts of economic, medical, demographic, historical, and scientific data if you look hard enough.

The key to finding these data sets is to track down the relevant papers. There is an academic literature on just about any topic of interest. Google Scholar is the most accessible source of research publications. Search by topic, and perhaps "Open Science" or "data." Research publications will typically provide pointers to where its associated data can be found. If not, contacting the author directly with a request should quickly yield the desired result.

The biggest catch with using published data sets is that someone else has worked hard to analyze them before you got to them, so these previously mined sources may have been sucked dry of interesting new results. But bringing fresh questions to old data generally opens new possibilities.

Often interesting data science projects involve collaborations between researchers from different disciplines, such as the social and natural sciences. These people speak different languages than you do, and may seem intimidating at first. But they often welcome collaboration, and once you get past the jargon it is usually possible to understand their issues on a reasonable level without specialized study. Be assured that people from other disciplines are generally not any smarter than you are.

Sweat Equity

Sometimes you will have to work for your data, instead of just taking it from others. Much historical data still exists only in books or other paper documents, thus requiring manual entry and curation. A graph or table might contain information that we need, but it can be hard to get numbers from a graphic locked in a PDF (portable document format) file.

I have observed that computationally-oriented people vastly over-estimate the amount of effort it takes to do manual data entry. At one record per minute, you can easily enter 1,000 records in only two work days. Instead, computational people tend to devote massive efforts trying to avoid such grunt work, like hunting in vain for optical character recognition (OCR) systems that don't make

a mess of the file, or spending more time cleaning up a noisy scan than it would take to just type it in again fresh.

A middle ground here comes in paying someone else to do the dirty work for you. Crowdsourcing platforms like Amazon Turk and CrowdFlower enable you to pay for armies of people to help you extract data, or even collect it in the first place. Tasks requiring human annotation like labeling images or answering surveys are particularly good use of remote workers. Crowdsourcing will be discussed in greater detail in Section 3.5.

Many amazing open data resources have been built up by teams of contributors, like Wikipedia, Freebase, and IMDb. But there is an important concept to remember: people generally work better when you pay them.

3.2.2 Scraping

Webpages often contain valuable text and numerical data, which we would like to get our hands on. For example, in our project to build a gambling system for the sport of jai-alai, we needed to feed our system the results of yesterday's matches and the schedule of what games were going on today. Our solution was to scrape the websites of jai-alai betting establishments, which posted this information for their fans.

There are two distinct steps to make this happen, spidering and scraping:

- *Spidering* is the process of downloading the right set of pages for analysis.
- *Scraping* is the fine art of stripping this content from each page to prepare it for computational analysis.

The first thing to realize is that webpages are generally written in simple-to-understand formatting languages like HTML and/or JavaScript. Your browser knows these languages, and interprets the text of the webpage as a program to specify what to display. By calling a function that emulates/pretends to be a web browser, your program can download any webpage and interpret the contents for analysis.

Traditionally, scraping programs were site-specific scripts hacked up to look for particular HTML patterns flanking the content of interest. This exploited the fact that large numbers of pages on specific websites are generated by programs themselves, and hence highly predictable in their format. But such scripts tend to be ugly and brittle, breaking whenever the target website tinkers with the internal structure of its pages.

Today, libraries in languages like Python (see BeautifulSoup) make it easier to write robust spiders and scrapers. Indeed, someone else probably has *already* written a spider/scraper for every popular website and made it available on SourceForge or Github, so search before you code.

Certain spidering missions may be trivial, for example, hitting a single URL (uniform resource locator) at regular time intervals. Such patterns occur in monitoring, say, the sales rank of this book from its Amazon page. Somewhat more sophisticated approaches to spidering are based on the name regularity of the

underlying URLs. If all the pages on a site are specified by the date or product ID number, for example `http://www.amazon.com/gp/product/1107041376/`, iterating through the entire range of interesting values becomes just a matter of counting.

The most advanced form of spidering is *web crawling*, where you systematically traverse all outgoing links from a given root page, continuing recursively until you have visited every page on the target website. This is what Google does in indexing the web. You can do it too, with enough patience and easy-to-find web crawling libraries in Python.

Please understand that politeness limits how rapidly you should spider/crawl a given website. It is considered bad form to hit a site more than once a second, and indeed best practices dictate that providers block access to the people who are hammering them.

Every major website contains a *terms of service* document that restricts what you can legally do with any associated data. Generally speaking, most sites will leave you alone provided you don't hammer them, and do not redistribute any data you scrape. Understand that this is an observation, not a legal opinion. Indeed, read about the Aaron Schwartz case, where a well-known Internet figure was brought up on serious criminal charges for violating terms of services in spidering/scraping journal articles, and literally hounded to death. If you are attempting a web-scraping project professionally, be sure that management understands the terms of service before you get too creative with someone else's property.

3.2.3 Logging

If you own a potential data source, treat it like you own it. Internal access to a web service, communications device, or laboratory instrument grants you the right and responsibility to log all activity for downstream analysis.

Amazing things can be done with ambient data collection from weblogs and sensing devices, soon destined to explode with the coming "Internet of Things." The accelerometers in cell phones can be used to measure the strength of earthquakes, with the correlation of events within a region sufficient to filter out people driving on bumpy roads or leaving their phones in a clothes dryer. Monitoring the GPS data of a fleet of taxi cabs tracks traffic congestion on city streets. Computational analysis of image and video streams opens the door to countless applications. Another cool idea is to use cameras as weather instruments, by looking at the color of the sky in the background of the millions of photographs uploaded to photo sites daily.

The primary reason to instrument your system to collect data is because you can. You might not know exactly what to do with it now, but any well-constructed data set is likely to become of value once it hits a certain critical mass of size.

Current storage costs make clear just how low a barrier it is to instrument a system. My local Costco is currently selling three terabyte disk drive for under $100, which is Big O of nothing. If each transaction record takes 1 kilobyte (one

thousand characters), this device in principle has room for 3 billion records, roughly one for every two people on earth.

The important considerations in designing any logging system are:

- Build it to endure with limited maintenance. Set it and forget it, by provisioning it with enough storage for unlimited expansion, and a backup.
- Store all fields of possible value, without going crazy.
- Use a human-readable format or transactions database, so you can understand exactly what is in there when the time comes, months or years later, to sit down and analyze your data.

3.3 Cleaning Data

"Garbage in, garbage out" is the fundamental principle of data analysis. The road from raw data to a clean, analyzable data set can be a long one.

Many potential issues can arise in cleaning data for analysis. In this section, we discuss identifying processing artifacts and integrating diverse data sets. Our focus here is the processing *before* we do our real analysis, to make sure that the garbage never gets in in the first place.

> *Take-Home Lesson*: Savvy painting restorers only do things to the original that are reversible. They never do harm. Similarly, data cleaning is always done on a copy of the original data, ideally by a pipeline that makes changes in a systematic and repeatable way.

3.3.1 Errors vs. Artifacts

Under ancient Jewish law, if a suspect on trial was unanimously found guilty by all judges, then this suspect would be *acquitted*. The judges had noticed that unanimous agreement often indicates the presence of a systemic error in the judicial process. They reasoned that when something seems too good to be true, a mistake has likely been made somewhere.

If we view data items as measurements about some aspect of the world, data *errors* represent information that is fundamentally lost in acquisition. The Gaussian noise blurring the resolution of our sensors represents error, precision which has been permanently lost. The two hours of missing logs because the server crashed represents data error: it is information which cannot be reconstructed again.

By contrast, *artifacts* are generally systematic problems arising from processing done to the raw information it was constructed from. The good news is that processing artifacts can be corrected, so long as the original raw data set remains available. The bad news is that these artifacts must be detected before they can be corrected.

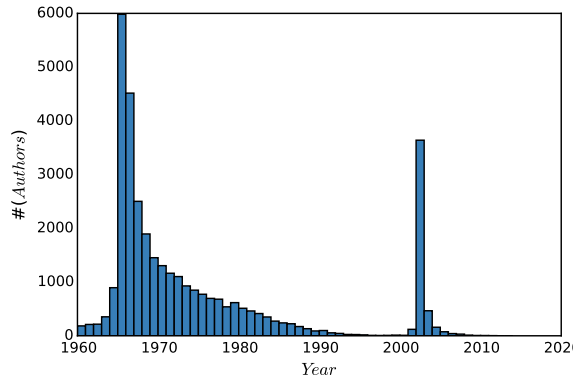


Figure 3.2: What artifacts can you find in this time series, counting the number of author's names first appearing in the scientific literature each year?

The key to detecting processing artifacts is the "sniff test," examining the product closely enough to get a whiff of something bad. Something bad is usually something unexpected or surprising, because people are naturally optimists. Surprising observations are what data scientists live for. Indeed, such insights are the primary reason we do what we do. But in my experience, most surprises turn out to be artifacts, so we must look at them skeptically.

Figure 3.2 presents computational results from a project where we investigated the process of scientific publication. It shows a time series of the 100,000 most prolific authors, binned according to the year of their first paper appearing in Pubmed, an essentially complete bibliography of the biomedical literature.

Study this figure closely, and see if you can discover any artifacts worth commenting on. I see at least two of them. Extra credit will be awarded if you can figure out what caused the problem.

The key to finding artifacts is to look for anomalies in the data, that contradict what you expect to see. What *should* the distribution in the number of virgin authors look like, and how should it change over time? First, construct a prior distribution of what you expect to see, so that you can then properly evaluate potential anomalies against it.

My intuition says that the distribution of new top scientists should be pretty flat, because new stars are born with every successive class of graduate students. I would also guess that there may be a gradual drift upward as population expands, and more people enter the scientific community. But that's not what I see in Figure 3.2. So try to enumerate what the anomalies/potential artifacts are...

I see two big bumps when I look at Figure 3.2: a left bump starting around

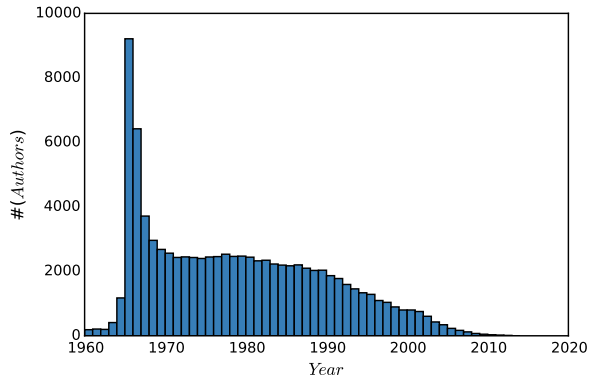


Figure 3.3: The cleaned data removes these artifacts, and the resulting distribution looks correct.

1965, and a peak which explodes in 2002. On reflection, the leftmost bump makes sense. This left peak occurs the year when Pubmed first started to systematically collect bibliographic records. Although there is some very incomplete data from 1960–1964, most older scientists who had been publishing papers for several years would "emerge" only with the start of systematic records in 1965. So this explains the left peak, which then settles down by 1970 to what looks like the flat distribution we expected.

But what about that giant 2002 peak? And the decline in new authors to almost zero in the years which precede it? A similar decline is also visible to the right of the big peak. Were all the world's major scientists destined to be born in 2002?

A careful inspection of the records in the big peak revealed the source of the anomaly: first names. In the early days of Pubmed, authors were identified by their initials and last names. But late in 2001, *SS Skiena* became *Steven S. Skiena*, so it *looked* like a new author emerging from the heavens.

But why the declines to nothingness to the left and right of this peak? Recall that we limited this study to the 100,000 most prolific scientists. A scientific rock star emerging in 1998 would be unlikely to appear in this ranking because their name was doomed to change a few years later, not leaving enough time to accumulate a full career of papers. Similar things happen at the very right of the distribution: newly created scientists in 2010 would never be able to achieve a full career's work in only a couple of years. Both phenomena are neatly explained by this first name basis.

Cleaning this data to unify name references took us a few iterations to get right. Even after eliminating the 2002 peak, we still saw a substantial dip in prominent scientists starting their careers in the mid 1990s. This was because many people who had a great half career pre-first names and a second great half career post-first names did not rise to the threshold of a great full career

in either single period. Thus we had to match all the names in the full *before* identifying who were the top 100,000 scientists.

Figure 3.3 shows our final distribution of authors, which matches the platonic ideal of what we expected the distribution to be. Don't be too quick to rationalize away how your data looks coming out of the computer. My collaborators were at one point ready to write off the 2002 bump as due to increases in research funding or the creation of new scientific journals. Always be suspicious of whether your data is clean enough to trust.

3.3.2 Data Compatibility

We say that a comparison of two items is "apples to apples" when it is fair comparison, that the items involved are similar enough that they can be meaningfully stood up against each other. In contrast, "apples to oranges" comparisons are ultimately meaningless. For example:

- It makes no sense to compare weights of 123.5 against 78.9, when one is in pounds and the other is in kilograms.
- It makes no sense to directly compare the movie gross of *Gone with the Wind* against that of *Avatar*, because 1939 dollars are 15.43 times more valuable than 2009 dollars.
- It makes no sense to compare the price of gold at noon today in New York and London, because the time zones are five hours off, and the prices affected by intervening events.
- It makes no sense to compare the stock price of Microsoft on February 17, 2003 to that of February 18, 2003, because the intervening 2-for-1 stock split cut the price in half, but reflects no change in real value.

These types of data comparability issues arise whenever data sets are merged. Here I hope to show you how insidious such comparability issues can be, to sensitize you as to why you need to be aware of them. Further, for certain important classes of conversions I point to ways to deal with them.

> *Take-Home Lesson*: Review the meaning of each of the fields in any data set you work with. If you do not understand what's in there down to the units of measurement, there is no sensible way you can use it.

Unit Conversions

Quantifying observations in physical systems requires standard units of measurement. Unfortunately there exist many functionally equivalent but incompatible systems of measurement. My 12-year old daughter and I both weigh about 70, but one of us is in pounds and the other in kilograms.

Disastrous things like rocket explosions happen when measurements are entered into computer systems using the wrong units of measurement. In particular, NASA lost the $125 million Mars Climate Orbiter space mission on September 23, 1999 due to a metric-to-English conversion issue.

Such problems are best addressed by selecting a single system of measurements and sticking to it. The metric system offers several advantages over the traditional English system. In particular, individual measurements are naturally expressed as single decimal quantities (like 3.28 meters) instead of incomparable pairs of quantities (5 feet, 8 inches). This same issue arises in measuring angles (radians vs. degrees/seconds) and weight (kilograms vs. pounds/oz).

Sticking to the metric system does not by itself solve all comparability issues, since there is nothing to prevent you from mixing heights in meters and centimeters. But it is a good start.

How can you defend yourself against incompatible units when merging data sets? Vigilance has to be your main weapon. Make sure that you know the intended units for each numerical column in your data set, and verify compatibility when merging. Any column which does not have an associated unit or object type should immediately be suspect.

When merging records from diverse sources, it is an excellent practice to create a new "origin" or "source" field to identify where each record came from. This provides at least the hope that unit conversion mistakes can be corrected later, by systematically operating on the records from the problematic source.

A partially-automated procedure to detect such problems can be devised from statistical significance testing, to be discussed in Section 5.3. Suppose we were to plot the frequencies of human heights in a merged data set of English (feet) and metric (meter) measurements. We would see one peak in the distribution around 1.8 and a second around 5.5. The existence of multiple peaks in a distribution should make us suspicious. The p-value resulting from significance testing on the two input populations provides a rigorous measurement of the degree to which our suspicions are validated.

Numerical Representation Conversions

Numerical features are the easiest to incorporate into mathematical models. Indeed, certain machine learning algorithms such as linear regression and support vector machines work only with numerically-coded data. But even turning numbers into numbers can be a subtle problem. Numerical fields might be represented in different ways: as integers (123), as decimals (123.5), or even as fractions (123 1/2). Numbers can even be represented as text, requiring the conversion from "ten million" to 10000000 for numerical processing.

Numerical representation issues can take credit for destroying another rocket ship. An Ariane 5 rocket launched at a cost of $500 million on June 4, 1996 exploded forty seconds after lift-off, with the cause ultimately ascribed to an unsuccessful conversion of a 64-bit floating point number to a 16-bit integer.

The distinction between integers and floating point (real) numbers is important to maintain. Integers are counting numbers: quantities which are really

discrete should be represented as integers. Physically measured quantities are never precisely quantified, because we live in a continuous world. Thus all measurements should be reported as real numbers. Integer approximations of real numbers are sometimes used in a misbegotten attempt to save space. Don't do this: the quantification effects of rounding or truncation introduces artifacts.

In one particularly clumsy data set we encountered, baby weights were represented as two integer fields (pounds and the remaining ounces). Much better would have been to combine them into a single decimal quantity.

Name Unification

Integrating records from two distinct data sets requires them to share a common key field. Names are frequently used as key fields, but they are often reported inconsistently. Is *José* the same fellow as *Jose*? Such diacritic marks are banned from the official birth records of several U.S. states, in an aggressive attempt to force them to be consistent.

As another case in point, databases show my publications as authored by the Cartesian product of my first (*Steve*, *Steven*, or *S.*), middle (*Sol*, *S.*, or blank), and last (*Skiena*) names, allowing for nine different variations. And things get worse if we include misspellings. I can find myself on Google with a first name of *Stephen* and last names of *Skienna* and *Skeina*.

Unifying records by key is a very ugly problem, which doesn't have a magic bullet. This is exactly why ID numbers were invented, so use them as keys if you possibly can.

The best general technique is unification: doing simple text transformations to reduce each name to a single canonical version. Converting all strings to lower case increases the number of (usually correct) collisions. Eliminating middle names or at least reducing them to an abbreviation creates even more name matches/collisions, as does mapping first names to canonical versions (like turning all *Steve*s into *Steven*s).

Any such transformation runs the risk of creating Frankenstein-people, single records assembled from multiple bodies. Applications differ in whether the greater danger lies in merging too aggressively or too timidly. Figure out where your task sits on this spectrum and act accordingly.

An important concern in merging data sets is *character code* unification. Characters in text strings are assigned numerical representations, with the mapping between symbols and number governed by the character code standard. Unfortunately, there are several different character code standards in common usage, meaning that what you scrape from a webpage might not be in the same character code as assumed by the system which will process it.

Historically, the good old 7-bit *ASCII* code standard was expanded to the 8-bit *IS0 8859-1 Latin* alphabet code, which adds characters and punctuation marks from several European languages. *UTF-8* is an encoding of all Unicode characters using variable numbers of 8-bit blocks, which is backwards compatible with ASCII. It is the dominant encoding for web-pages, although other systems remain in use.

Correctly unifying character codes after merging is pretty much impossible. You must have the discipline to pick a single code as a standard, and check the encoding of each input file on preprocessing, converting it to the target before further work.

Time/Date Unification

Data/time stamps are used to infer the relative order of events, and group events by relative simultaneity. Integrating event data from multiple sources requires careful cleaning to ensure meaningful results.

First let us consider issues in measuring time. The clocks from two computers never exactly agree, so precisely aligning logs from different systems requires a mix of work and guesswork. There are also time zone issues when dealing with data from different regions, as well as diversities in local rules governing changes in daylight saving time.

The right answer here is to align all time measurements to *Coordinated Universal Time* (UTC), a modern standard subsuming the traditional *Greenwich Mean Time* (GMT). A related standard is *UNIX time*, which reports an event's precise time in terms of the number of elapsed seconds since 00:00:00 UTC on Thursday, January 1, 1970.

The Gregorian calendar is common throughout the technology world, although many other calendar systems are in use in different countries. Subtle algorithms must be used to convert between calendar systems, as described in [RD01]. A bigger problem for date alignment concerns the proper interpretation of time zones and the international date line.

Time series unification is often complicated by the nature of the business calendar. Financial markets are closed on weekends and holidays, making for questions of interpretation when you are correlating, say, stock prices to local temperature. What is the right moment over the weekend to measure temperature, so as to be consistent with other days of the week? Languages like Python contain extensive libraries to deal with financial time series data to get issues like this correct. Similar issues arise with monthly data, because months (and even years) have different lengths.

Financial Unification

Money makes the world go round, which is why so many data science projects revolve around financial time series. But money can be dirty, so this data requires cleaning.

One issue here is *currency conversion*, representing international prices using a standardized financial unit. Currency exchange rates can vary by a few percent within a given day, so certain applications require time-sensitive conversions. Conversion rates are not truly standardized. Different markets will each have different rates and *spreads*, the gap between buying and selling prices that cover the cost of conversion.

The other important correction is for inflation. The *time value of money* implies that a dollar today is (generally) more valuable than a dollar a year from now, with interest rates providing the right way to discount future dollars. Inflation rates are estimated by tracking price changes over baskets of items, and provide a way to standardize the purchasing power of a dollar over time.

Using unadjusted prices in a model over non-trivial periods of time is just begging for trouble. A group of my students once got very excited by the strong correlation observed between stock prices and oil prices over a thirty-year period, and so tried to use stock prices in a commodity prediction model. But both goods were priced in dollars, without any adjustment as they inflated. The time series of prices of essentially *any* pair of items will correlate strongly over time when you do not correct for inflation.

In fact, the most meaningful way to represent price changes over time is probably not differences but *returns*, which normalize the difference by the initial price:

$$r_i = \frac{p_{i+1} - p_i}{p_i}$$

This is more analogous to a percentage change, with the advantage here that taking the logarithm of this ratio becomes symmetric to gains and losses.

Financial time series contain many other subtleties which require cleaning. Many stocks give scheduled *dividends* to the shareholder on a particular date every year. Say, for example, that Microsoft will pay a $2.50 dividend on January 16. If you own a share of Microsoft at the start of business that day, you receive this check, so the value of the share then immediately drops by $2.50 the moment after the dividend is issued. This price decline reflects no real loss to the shareholder, but properly cleaned data needs to factor the dividend into the price of the stock. It is easy to imagine a model trained on uncorrected price data learning to sell stocks just prior to its issuing dividends, and feeling unjustly proud of itself for doing so.

3.3.3 Dealing with Missing Values

Not all data sets are complete. An important aspect of data cleaning is identifying fields for which data isn't there, and then properly compensating for them:

- What is the year of death of a living person?
- What should you do with a survey question left blank, or filled with an obviously outlandish value?
- What is the relative frequency of events too rare to see in a limited-size sample?

Numerical data sets expect a value for every element in a matrix. Setting missing values to zero is tempting, but generally wrong, because there is always some ambiguity as to whether these values should be interpreted as data or not.

Is someone's salary zero because he is unemployed, or did he just not answer the question?

The danger with using nonsense values as not-data symbols is that they can get misinterpreted as data when it comes time to build models. A linear regression model trained to predict salaries from age, education, and gender will have trouble with people who refused to answer the question.

Using a value like -1 as a no-data symbol has exactly the same deficiencies as zero. Indeed, be like the mathematician who is afraid of negative numbers: stop at nothing to avoid them.

> *Take-Home Lesson*: Separately maintain both the raw data and its cleaned version. The raw data is the ground truth, and must be preserved intact for future analysis. The cleaned data may be improved using imputation to fill in missing values. But keep raw data distinct from cleaned, so we can investigate different approaches to guessing.

So how should we deal with missing values? The simplest approach is to drop all records containing missing values. This works just fine when it leaves enough training data, provided the missing values are absent for non-systematic reasons. If the people refusing to state their salary were generally those above the mean, dropping these records will lead to biased results.

But typically we want to make use of records with missing fields. It can be better to estimate or *impute* missing values, instead of leaving them blank. We need general methods for filling in missing values. Candidates include:

- *Heuristic-based imputation*: Given sufficient knowledge of the underlying domain, we should be able to make a reasonable guess for the value of certain fields. If I need to fill in a value for the year you will die, guessing *birth year+80* will prove about right on average, and a lot faster than waiting for the final answer.

- *Mean value imputation*: Using the mean value of a variable as a proxy for missing values is generally sensible. First, adding more values with the mean leaves the mean unchanged, so we do not bias our statistics by such imputation. Second, fields with mean values add a vanilla flavor to most models, so they have a muted impact on any forecast made using the data.

  But the mean might not be appropriate if there is a systematic reason for missing data. Suppose we used the mean death-year in Wikipedia to impute the missing value for all living people. This would prove disastrous, with many people recorded as dying before they were actually born.

- *Random value imputation*: Another approach is to select a random value from the column to replace the missing value. This would seem to set us up for potentially lousy guesses, but that is actually the point. Repeatedly selecting random values permits statistical evaluation of the impact of imputation. If we run the model ten times with ten different imputed