CHAPTER 1

Lessons Learned from Software Analytics In Practice

In this chapter, we share our experience and views on software data analytics in practice with a retrospect to our previous work. Over ten years of joint research projects with the industry, we have encountered similar data analytics patterns in diverse organizations and in different problem cases. We discuss these patterns following a 'software analytics' framework: problem identification, data collection, descriptive statistics and decision making. We motivate the discussion by building our arguments and concepts around our experiences of the research process in six different industry research projects in four different organizations.

Keywords: *software analytics framework, industry research projects, data extraction, descriptive statistics, predictive analytics, prescriptive analytics*

Methods explained in this chapter: *Spearman rank correlation, Pearson correlation, Kolmogorov-Smirnov test, Chi-square goodness of fit test, t-test, Mann Whitney U-test, Kruskal-Wallis analysis of variance, k-nearest neighbour, linear regression, logistic regression, Naive Bayes, Neural Networks, Decision trees, ensembles, nearest neighbour sampling, feature selection, normalization*

1.1 Introduction

Modern organizations operate in an environment shaped by the advances of information and communication technologies. The economic growth in highly populated countries such as China, India, and Brazil force organizations from the traditionally strong economies of Europe, North America and Japan to globally compete for resources as well as markets for their goods and services. This poses an unprecedented need for the organizations to be efficient and agile [1]. Meanwhile, todays ubiquitous communication and computing technologies generate vast amounts of data in many forms and many magnitudes of what was available even a decade ago. The ability for organizations of all sizes to gain actionable insights from such data sources, not only to optimize their use of limited resources [2], but also to help shape their future, is now the key to survival. Those who can beat their competitors at the effectiveness with which they handle their informational resources are the ones likely to go beyond survival and enjoy sustained competitiveness. Software development organizations also follow the same trend.

Most of the management decisions in software engineering are based on the perceptions of the people about the state of the software and their estimations about its future states. Some of these decisions are resource allocation, team building, budget estimation and release planning. As the complexity of software systems and the interactions between increasing number of developers have grown, the need for a data driven decision making has emerged to solve common problems of the domain, such as completing projects on time, within budget and with minimum errors. Software projects are inherently difficult to control. Managers struggle to make many decisions under a lot of uncertainty. They would like to be confident on the product, team, and the processes. There are also many blind spots that may cause severe problems at any point within the project development life cycle. These concerns have drawn much attention to software measurement, software quality, and software cost/ effort estimation, namely descriptive and predictive analytics.

Data science is vital for software development organizations due to a paradigm shift around many kinds of data within development teams. We essentially need to have intuition on data, and this is just not statistics (statistical modelling, fitting, simulation) that we have learnt in school. Data science consists of analytics to use data to understand past and present (descriptive analytics), analyze past performance (predictive analytics), and to use optimization techniques (prescriptive analytics).

Software analytics is one of those unique fields that lies at the intersection of academia and practice. Software analytics research is empirically driven. Unlike traditional research methodologies, researchers study and learn from the data to build useful analytics. To produce insightful results for industrial practice, researchers need to use industrial data and build analytics.

Software analytics must follow a process that starts with problem identification, i.e., framing the business problem and the analytics problem. Throughout this process, stakeholder agreement needs to be obtained and/ or re-obtained through effective communication. The end goal for any software analytics project should be to address a genuine problem in the industry. Therefore, the outcome of the analytics project could be transferred and embedded into the decision making process of the organization. Sometimes, software analytics provides additional insights that stakeholders do not plan or imagine about it, and in such cases, the results could influence more than one phase in a software development process.

Data collection in software organizations is a complicated procedure. It requires identification and prioritization of data needs and available sources depending on the context. Qualitative and quantitative data collection methods are different depending on the problem. In some cases data acquisition requires a tool development first, followed by harmonizing, rescaling, cleaning and sharing data within the organization. In data collection section, we will cover potential complications and data scaling issues.

The simplest way to get insight from data is through descriptive statistics. Even a simple analysis or visualization may reveal hidden facts in the data. Later, predictive models utilizing data mining techniques may be built to aid decision making process in an organization. Selection of the suitable data mining techniques depending on the problem is critical while building the predictive model. Factors such as clarity of the problem, maturity of data collection process in a company, and expected benefits from the predictive model may also affect model construction. In addition, insights gained from descriptive statistics may be used in the construction of predictive models. Finally, new predictive models could be evaluated, which requires the definition of certain performance measures, appropriate statistical tests as well as the effect size analysis.

In our software analytics projects, we have used a methodology adapted from a typical data mining process (problem definition, data gathering and preparation, model building and evaluation, knowedge deployment [3]) to define the main steps we performed. Figure 1.1 depicts five main phases, the results of which could be used directly by practitioners, or could be linked to consecutive phases. The whole process does not necessarily end with a single cycle, but it is preferred to conduct several iterations in a software analytics project to make adjustments depending on the outcome of each phase.

In this chapter, we define each of these five phases in Figure 1.1 based on our experience in state-of-the-art case studies with various software organizations. We provide example techniques, tools, charts that we used to collect, analyze and model data, as well as to optimize and deploy the proposed model in different industry settings in order to solve two main challenges: software defect prediction and effort estimation. In the following subsections, we define the objectives of each phase in a software analytics project, share our experience on how to conduct each phase, and provide practical tips to address potential issues. All examples and lessons learned have come from our empirical studies rigorously conducted with industry partners. It is important to note that we do not provide the details of our past projects (e.g. reasons behind using a particular algorithm or a new metric, pseudo codes for model building, all performance measures, etc.) in this chapter due to space limitations. We suggest the reader to refer to our articles listed at the end of this chapter for such information.

1.2 Problem Selection

Problem selection in research differs depending on the nature of research. In natural and social sciences, the researcher/ scientist chooses a topic that he/ she is simply curious about. In natural sciences, the researcher aims to understand the natural phenomena to build the theoretical basis for prediction and this becomes the basis for invention or engineering. Whereas in social sciences, the researcher aims to understand human and societal phenomena to build the theoretical basis for prediction and intervention, and this becomes the basis

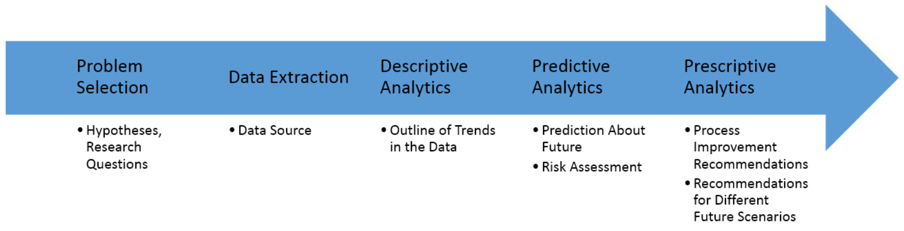
Problem Selection
• Hypotheses, Research Questions
Data Extraction
• Data Source
Descriptive Analytics
• Outline of Trends in the Data
Predictive Analytics
• Prediction About Future
• Risk Assessment
Prescriptive Analytics
• Process Improvement Recommendations
• Recommendations for Different Future Scenarios

Figure 1.1: Our Methodology in Software Analytics Projects

for prediction, basis for intervening and changing the world to create therapy, education, policy, motivation, etc. Both have rigorous experimental basis. In natural sciences, theories have to be testable, testing is done in a physical world which provides hard constraints on theories. In social sciences, on the other hand, theories have to be testable, testing is done in a behavioural world which provides probabilistic constraints.

Empirical software engineering, compared to natural sciences and social sciences, is still an immature field. It suffers from lack of understanding on empirical problems, issues, possibilities and research designs. Software engineering is the study and practice of building software systems. It is an experimental field that uses various forms of experiments to test the theory (i.e. requirements) and its model (i.e. implementation). These experiments include independent and dependent variables, manipulations, data collection and analysis.

Software engineering is a very rich field where we have systems that execute programs and processes, people who design and use machines, people who use and execute processes. This richness enables software engineering research and practice to cross cut to other disciplines such as: anthropology to understand families of systems, sociology to understand systems in context (i.e. relationships among systems as centralized, distribute, networked, etc.), social psychology to understand individual systems (i.e. component interaction), and personal psychology to understand individual system characteristics. People and processes can also be mapped to these disciplines: anthropology to understand projects and disciplines, sociology to understand interactions among teams or projects, social psychology to understand interactions of people and technology, and personal psychology to understand traits, dispositions of developers and managers. These intersections enable researchers to observe and abstract specific parts of the world and create a theory. From that theory they can create a usable model to represent that theory. It then becomes an iterative process to adjust both the theory and the model as they evolve. When the researcher is satisfied he/ she injects the model into the world. Injecting the model into the world changes the world and this leads to adjustments and extensions to the original theory. This then leads to further changes in the model and the world.

A good question to conduct a research is the one that is testable with the materials at hand. A good question in empirical software engineering research is not only the one that the researcher can conduct an experiment in a pragmatic manner, it is also a relevant one that addresses a genuine issue in the daily life of a software organization. Everyday many decisions need to be made in software organizations, such as determining what users want/ need, assessing architecture and design choices, evaluating functional characteristics or non-functional properties, evaluating and comparing technologies (i.e. supporting tools, product support, process support), determining what went wrong, and allocating resources effectively. Some of these challenges have also been addressed by process improvement (e.g. [4]) and data management models (e.g. [5]).

In many software development organizations, there is little evidence to inform decisions. Companies do not know exception cases or fundamental mechanisms of software tools, methods and techniques, and processes. Therefore empirical studies are the key to show these mechanisms and eliminate alternative explanations. An empirical study is a study that reconciles theory and reality. In order to conduct better empirical studies, researchers need to establish principles that are causal (correlated, testable theory), actionable and general. They need to answer important questions rather than focusing on a nice solution and trying to map that solution to a problem that does not exist.

In problem selection step, we need to ask the right or important questions. It is important to involve the domain expert/ practitioner at this stage. We may ask different types of questions: existence, description/ classification, composition, relationships, descriptive-comparative, causality, or causality-comparison interactions. We need to make sure that the hypotheses are consistent with research questions since these will be the basis of the statement of problem. Failing to ask the right questions and inconsistencies in hypothesis formulation also would lead to methodological pitfalls such as hypothesis testing errors, issues of statistical power, issues in construction of dependent/ independent variables, reliability and validity issues.

Research, especially in software engineering, is all about balancing originality and relevance. Therefore, the researcher, before he/ she starts the research, should always ask this question What is the relevance of the outcome of this research for the practice?. In a previous case study, we jointly identified the problem with software development team during the project kick-off meeting [6]. The development team initially listed their business objectives (improve code quality, decrease defect rates, and measure time to fix defects), while the researchers aligned the goals of our project (building code measurement and version control repository, and defect prediction model) respectively. To achieve these goals, the roles and responsibilities were defined for both sides (practitioners and researchers); the expectations and potential output of the analytics project were discussed and agreed. Furthermore, it was decided that any output produced during the project (e.g. metrics from data extraction, charts from descriptive analytics) would be assessed together before moving to the next step.

In some cases, the business need may be defined too general or ambiguous, and therefore, the researchers should conduct further analysis to identify and frame the analytics problem. For instance, in a case study with a large scale software organization, the team leaders initially defined their problem as "measuring release readiness of a software product" [7]. The problem was too general for researchers, since it may indicate building an analytics for a) measuring the "readiness" of a software release in terms of budget, schedule, or pre-release defect rates, or for b) deciding which features could be deployed in the next release, or for c) measuring the reliability of a software release in terms of residual (post-release) defects. Through interviews with junior to senior developers and team leaders, the business need was re-defined as "assessing the final reliability of a software product prior to release" in order to decide how much resources would be allocated for maintenance (bug fixing) activities. We took this business need and transformed into an analytics problem as follows: Building a model that would estimate residual (post-release) defect density of software product *at any time during a release.* At any given time during a release, the model could learn from the metrics that were available, and predict the residual defects in software product. To do that, we further decided on the input of the model as software metrics from requirements, design, development and testing phases. The details of the model could be found in [7].

1.3 Data Collection

In our software analytics projects, we initially design the dataset required for a particular problem. Afterwards, we extract the data based on our initial data design through quantitative and qualitative techniques. In this section we describe these steps with guidelines to the practitioners based on our past experience.

1.3.1 Datasets

Datasets for Predictive Analytics

Most organizations want to predict the number of defects or identify defect-prone modules in software systems, before they are deployed. Numerous statistical methods and Artificial Intelligence (AI) techniques have been employed in order to predict defects that a software system will reveal in operation or during testing.

As we indicated earlier, research in software engineering should focus on the relevance of the outcome for the practice. Our goal has so far been to find solutions to the problems of industry by improving software development life cycle and hence software quality. We have used predictive analytics to catch defect for large-scale companies and Small Medium Enterprises (SMEs) specialized in various domains such as telecommunications, ERP, banking systems and embedded systems for white-good manufacturing. We collected data from industry and used these datasets in our research in addition to publicly available datasets such as MDP repository for NASA datasets and PROMISE data repository [8]. Moreover, we donated all datasets we collected from industry to PROMISE data repository.

In our previous industry collaborations, we mostly use static code metrics to build predictive analytics. *Static code metrics* consist of McCabe, LOC (Lines of Code), Halstead, and Chidamber and Kemerer (CK) Object-Oriented metrics as shown in Table 1.1.

McCabe metrics, which are based on a graph theoretic approach, provide a quantitative basis to estimate the code complexity based on the decision structure of a program [9]. A program can be represented as a directed graph G where nodes are the program statements and directed edges represent the flow of control from one statement to another. The idea behind McCabe metrics is that the more structural complexity of a code gets, the more difficult it gets to test and maintain the code and hence the likelihood of defects increases. For instance, cyclomatic complexity $v(G)$ of a program with a corresponding directed graph G with e edges, n nodes and p connected components is the number of linearly independent paths through that program and it is calculated as $v(G) = e - n + p$. Therefore, if there is no branching in a module (i.e., no conditional statements such as if, while, etc.), then cyclomatic complexity $v(G)$ of that module is 1. For a module with a single conditional statement, $v(G)$ is 2. The rest of the McCabe metrics consist of cyclomatic complexity, cyclomatic density, decision density, essential density and maintenance severity. The description of these metrics and the relationship between each other are given in table 1.1.

LOC metrics are simple measures that can be extracted from the code. These include, but are not limited to total lines of code, blank lines of code, lines of commented code, lines of code and comment, and lines of executable code metrics. Description of these metrics are given in Table 1.1.

Halstead metrics, which are also listed in Table 1.1, measure a program module's complexity directly from source code, with emphasis on computational complexity [10]. These metrics were developed as a means of determining measure of complexity directly from the operators and operands in the module. The rationale behind Halstead metrics is that the harder the code to read, the more defect prone the modules are.

McCabe, LOC and Halstead metrics were developed with traditional methods in mind, hence they do not lend themselves to Object Oriented (OO) notions such as classes, inheritance, encapsulation and message passing. OO metrics were developed by Chidamber and Kemerer (i.e., CK OO metrics) in order to measure unique aspects of OO approach [11]. CK OO metrics are listed in Table 1.2 together with their definitions.

Some researchers criticized the use of static code metrics to learn defect predictors due to their limited content [12, 13]. However, static code metrics are easy to collect and interpret. In our early research, we used static code metrics to build defect prediction models for a local white goods manufacturing company [14] and for software companies specialized in ERP software and banking systems [15]. The prediction performance results we obtained were quite promising: $pd = 82$ %, $pf = 33$ %, $balance = 73$ % for the best case, and $pd = 82$ %, $pf = 47$ % $balance = 61$ %. These results are higher than results obtained for manual code reviews, which are currently used in industry. Moreover manual

Table 1.1: List of static code metrics

McCabe Metrics	
Attribute	***Description***
Cyclomatic complexity ($v(G)$)	Number of linearly independent paths
Cyclomatic density ($vd(G)$)	The ratio of the files cyclomatic complexity to its length
Decision density ($dd(G)$)	Condition/decision
Essential complexity ($ev(G)$)	The degree to which a file contains unstructured constructs
Essential density ($ed(G)$)	(ev(G) - 1)/(v(G) - 1)
Maintenance severity	ev(G)/v(G)
Lines of Code Metrics	
Attribute	***Description***
Total lines of code	Total number of lines on source code.
Blank lines of code	Total number of blank lines in source code.
Lines of commented code	Total number of lines consisting of code comments
Lines of code and comment	Total number of source code lines that include both executable statements and comments
Lines of executable code	Total number of the actual code statements that are executable
Halstead Metrics	
Attribute	***Description***
n1	Unique operands count
n2	Unique operators count
N1	Total operands count
N2	Total operators count
Level (L)	(2/n1)/(n2/N2)
Difficulty (D)	1/L
Length (N)	N1 + N2
Volume (V)	N * log(n)
Programming effort (E)	D * V
Programming time (T)	E/18

Table 1.2: List of CK Object Oriented metrics

Attribute	***Description***
Weighted methods per class (WMC)	Sum of the complexity of the methods in a class.
Depth of inheritance tree (DIT)	DIT for a class is the maximum length from the node to the root of the tree of class inheritance.
Number of Children (NOC)	Number of immediate subclasses subordinated in the class hierarchy.
Coupling between object classes (CBO)	Count of the number of other classes to which a class is coupled.
Response for a Class (RFC)	The union of the set of methods called by each method in a class.
Lack of cohesion in methods ($LCOM$)	The union of the set of instance variables used by each method in a class.

code inspections are quite labor intensive. Depending on the review methods, 8 to 20 LOC/minute can be inspected per reviewer and a review team mostly consists of four or six members [16].

In order to improve prediction performance, we assigned relevant weights to static code attributes according to their importance which improves defect prediction performance[17] and our proposed method yielded at least equivalent and in some cases better than the currently best defect predictor [18]. In another study, we reduced the probability of false alarms (pf) by supplementing static code metrics with a call graph based ranking (CGBR) framework [19, 20]. In one of our studies, where we also used CGBR framework in order to increase the information content of prediction models, defect prediction performance improved for large and complex systems, while for small systems, prediction models without CGBR framework achieved the same prediction performance [21].

During the research project we conducted for Turkey's largest GSM operator/telecommunications company, we built prediction models by employing 22 projects and 10 releases of one of company's major software products [6]. When the research project started, defects were not matched with files in issue management system. Developers could not allocate extra time to write all the defects they fixed during the testing phase because of their workload and other business priorities. In addition, matching those defects with the corresponding files of software could not be automatically handled. Therefore, the prediction models could not be trained using company data. Instead, similar projects from cross-company (CC) data were selected by using Nearest Neighbour (NN) sampling. Projects from MDP repository for NASA, which are representative of US software, were selected as cross-company data. In order to increase the information content, information extracted from SOA based call graphs (i.e., dependency data between modules) were used. Later the defect prediction model was deployed into company's software development process. In another research project, we also used Within Company Data (i.e., data from same company but different projects) in order to build defect predictors for embedded systems software for white-goods [22] . During this research project, we mixed WC data with CC data to train prediction models whenever Within Company Data (WC) data was limited. Complementing CC data with WC data yielded better performance results.

There is still considerable room for improvement of prediction performance (i.e., lower pfs and higher *pd*s). Researchers are actively looking for better code metrics which, potentially, will yield "better" predictors. For this purpose, in one of our research we used churn metrics as well as static code metrics and CGBR framework in order to build defect predictors for different defect categories [23]. According to the results we obtained, churn metrics performed the best for predicting all types of defects. Code churn is a measure of the amount of code change taking place within a software unit over time. Churn often propagates across dependencies. If a component C_1, which has dependencies with component C_2, changes (churns) a lot between versions, we expect component C_2 to undergo a certain amount of churn in order to keep in synch with component C_1. Together, a high degree of dependence plus churn can cause errors that will propagate through a system, reducing its reliability. A list of churn metrics are given in Table 1.3.

Table 1.3: List of churn metrics

Attribute	***Description***
$Commits$	Number of commits made for a file
$Committers$	Number of committers who committed a file
$CommitsLast$	Number of commits made for a file since last release
$CommittersLast$	Number of developers who committed a file since last release
$rmlLast$	Number of removed lines from a file since last release
$alLast$	Number of added lines to a file since last release
rml	Number of removed lines from a file
al	Number of added lines to a file
$TopDevPercent$	Percentage of top developers who committed a file

The above prediction models ignore the causal effects of programmers and designers on software defects. In other words, datasets used to learn such defect predictors consist of product-related (static code metrics) and *process*-related (churn) metrics, but not *people*-related metrics. On the other hand, people's thought processes have a significant impact on software quality as software is designed, implemented and tested by people. In the literature, various people-related metrics have been used to build defect predictors, yet these are not directly related to people's thought processes or their cognitive aspects [24, 25, 26, 27, 28].

In our research, we focussed on a specific human cognitive aspect, namely confirmation bias [29, 30, 31, 32]. Based on founded theories in cognitive psychology, they defined a set of metrics to quantify developers' confirmatory behaviour [33] and used these metrics to learn defect predictors [34]. Prediction performance of models that were learnt from these metrics were comparable with those of the models that were learnt from static code and churn metrics, respectively. Confirmatory behaviour is a single human aspect, yet the results obtained were quite promising. Our findings support the fact that we should study human aspects further to improve performance of defect prediction models.

In addition to individual metrics, metrics to quantify social interactions among software engineers serve as datasets having enhanced information content. In the literature, there are also other attempts where using social interaction networks to learn defect predictors yielded in promising performance results [35, 36, 37, 38]. In addition to these empirical studies, our research group also investigated social interaction among developers [39] and adapted various metrics from the complex network literature [40]. Some members of our research group also used social network metrics for defect prediction on two open source datasets namely, development data of IBM Rational, Team Concert (RTC) and Drupal [39]. The results revealed that compared to other set of metrics such as churn metrics using social network metrics on these datasets either considerably decreases high false alarm rates without compromising the detection rates or considerably increases low prediction rates without compromising low false alarm rates.

We have been building defect prediction models for large scale software development companies and SMEs specialized in domains such as ERP, finance, online banking,telecommunications and embedded systems. Based on our experience, we recommend the following roadmap to decide on the content of the dataset that will be collected to build defect prediction models:

1. **Start with static code metrics:** Although static code metrics (e.g., McCabe, LOC and Halstead metrics) have limited information content, they are easy to collect and use. They can be automatically and cheaply collected even for very large systems. Moreover, they give idea about the quality of the code and they can be used to decide which modules are worthy of manual inspections.

2. **Try to enhance defect predictors that are learnt from static code metrics:** Try methods such as weighting static code attributes or employing CGBR framework in order to improve prediction performance of the models that are built by using static code attributes.

3. **Do not give up on static code metrics even if you do not have defect data:** In some software development companies defects may not be stored during the development activities. There might even not exist a process to match the defects with the files in order to keep track of the reasons for any change in the software system. It will not be a feasible solution to match defects manually due to the heavy workload of developers. One possible way might be to call for emergency meetings with the software engineers and senior management in order to convince them change their existing code development process. As a change in development process, developers might be forced to checkin the source code to the version management system with a unique id of the test defect or requirement request. Referring to these unique IDs, one can identify which changes in the source code are intended to fix defects and which are for new requirement requests by referring to commit logs. Adaptation of the process change will take time, moreover collection of adequate defect data will also take some time. While defect data is being collected, in the meantime Cross Company (CC) data can be used to build defect predictors. Within Company (WC) data that has been collected so far can be used in combination with CC data until enough local data is collected. Methods mentioned in step 2 can be used with CC data and CC+WC data as well.

4. **Enhance defect prediction using churn metrics and/or social network metrics:** Churn metrics can be automatically collected from the logs of version management systems. Automated collection of social interaction metrics is also possible through mining issue management systems, version management systems and developers' emails, which they send each other to discuss software issues and new features. After extracting static

code, churn and social interaction metrics, defect prediction models can be built by employing all combinations of these three different metrics types (i.e., static code metrics; churn metrics; social interaction metrics; static code and churn metrics; static code and social interaction metrics; churn and social interaction metrics; and static code, churn and social interaction metrics). Based on the performance comparison results of these models, it can be decided which models will be used to identify defect prone parts of the software.

5. **Include metrics related to individual human aspects:** It is much more challenging to employ individual human metrics to build defect prediction models. Formation of the metrics set and defining a methodology to collect metrics values requires interdisciplinary research including fields such as cognitive and behavioural psychology besides traditional software engineering. Moreover, one needs to face the challenges of qualitative empirical studies which are mentioned in detail in section 1.3.2 titled "Data Extraction". Due to its challenges, we employed confirmation bias metrics to build prediction models at later stages during the field studies we conducted at software companies.

Datasets for Effort Estimation Models

While building models for effort estimation, we used both datasets, which we prepared by collecting data from various local software companies in Turkey, as well as publicly available datasets. Among the datasets we collected are SDR datasets [41, 42, 43, 44]. We used the COCOMO II Data Collection Questionnaire in order to collect SDR datasets. SDR datasets consist of 24 projects, which were implemented in 2000's. An exemplary dataset is shown in Table 1.4 in order to give an idea about content and format of the datasets we used for software effort estimation. Each row in Table 1.4 corresponds to a different project. These projects are represented by the nominal attributes from the COCOMO II model along with their size in terms of Lines of Code (LOC) and the actual effort spent for completing the projects. We also collected datasets from two large scale Turkish software companies, which are specialized in telecommunication and online banking domains, respectively [45, 46].

Publicly available datasets, which we used in our research projects were obtained from Promise Data Repository [8] and International Software Benchmarking Standards Group (ISBSG), which is a non-profit organization that maintains a software project management database from a variety of organizations [47].

Project managers can benefit from the learning based effort estimation models we have used in order to make accurate estimates when bidding for a new project or when allocating the resources among different software projects as well as among different phases of the software development lifecycle. Using our own datasets together with publicly available datasets helped us form cross domain datasets (i.e., datasets from different application domains) in addition to within domain datasets (i.e., datasets from a similar application domain). Our research results indicated that for embedded systems it is better to use cross domain datasets rather than using within domain datasets [48]. Based on our experience, we recommend practitioners to build effort estimation models by using projects from a wider spectrum as dataset, rather than using projects from a similar application. Analogy based models which are widely used for effort estimation, assume the availability of project data that are similar to the project at hand, which can be difficult to obtain. In other words, there may be no similar projects in house and obtaining data from other companies may delimited due to confidentiality. Our proposed framework, which uses makes it possible to use cross-domain data suggests that it is not necessary to take care of particular characteristics of a project (i.e., its similarity with other projects), while constructing effort estimation models [48].

Table 1.4: An example dataset for effort estimation

Project	Nominal attributes (as defined in COCOMO II)	LOC	Effort
P1	1.00,1.08,1.30,1.00,1.00,0.87,1.00,0.86,1.00,0.70,1.21,1.00,0.91,1.00,1.08	70	278
P2	1.40,1.08,1.15,1.30,1.21,1.00,1.00,0.71,0.82,0.70,1.00,0.95,0.91,0.91,1.08	227	1,181
P3	1.00,1.08,1.15,1.30,1.06,0.87,1.07,0.86,1.00,0.86,1.10,0.95,0.91,1.00,1.08	177.9	1,248
P4	1.15,0.94,1.15,1.00,1.00,0.87,0.87,1.00,1.00,1.00,1.00,0.95,0.91,1.00,1.08	115.8	480
P5	1.15,0.94,1.15,1.00,1.00,0.87,0.87,1.00,1.00,1.00,1.00,0.95,0.91,1.00,1.08	29.5	120
P6	1.15,0.94,1.15,1.00,1.00,0.87,0.87,1.00,1.00,1.00,1.00,0.95,0.91,1.00,1.08	19.7	60
P7	1.15,0.94,1.15,1.00,1.00,0.87,0.87,1.00,1.00,1.00,1.00,0.95,0.91,1.00,1.08	66.6	300
P8	1.15,0.94,1.15,1.00,1.00,0.87,0.87,1.00,1.00,1.00,1.00,0.95,0.91,1.00,1.08	5.5	18
P9	1.15,0.94,1.15,1.00,1.00,0.87,0.87,1.00,1.00,1.00,1.00,0.95,0.91,1.00,1.08	10.4	50
P10	1.15,0.94,1.15,1.00,1.00,0.87,0.87,1.00,1.00,1.00,1.00,0.95,0.91,1.00,1.08	14.	60
P11	1.00,0.00,1.15,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00	16	114
P12	1.15,0.00,1.15,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00	6.5	42
P13	1.00,0.00,1.15,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00	13	60
P14	1.00,0.00,1.15,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00	8	42

1.3.2 Data Extraction

The validity of the obtained results in software analytics projects is limited by the quality of the extracted data. For this reason, accurate data extraction is one of the most important phases in the software analytics projects. In every research project, it is important to design the data extraction phase carefully.

The first step during data extraction is the choice of the right requirements for the data that will be extracted. If data requirements are changed after the data extraction phase, all the results of the study might be changed. Similarly, rescheduling extensions to surveys done with the industry might be time consuming.

Automation of the data extraction steps when possible, reduces the extraction effort in the long term. On the other hand, tracking the history of the data extraction scripts helps in repeating the older experiments when necessary. Over the years, we have employed several techniques for quantitative and qualitative data extraction for a given problem [49, 50]. In certain scenarios, we partly automated the qualitative techniques for efficiency. In this section, we describe these techniques and how we customized them for our problems.

Quantitative Data Extraction

Quantitative data can be either continuous data or discrete. Quantitative data can be extracted from different software artifacts including the source code and issue repositories. Similarly, quantitative techniques can be applied to post-process the qualitative data extracted in survey and questionnaire.

Source code repositories are arguably the primary data sources in software analytics projects. Every non-trivial software is stored and tracked in source code repositories. Source code repositories can be used to extract the state of a project at a given time snapshot or track the evolution of the software project over time. Evolution of the software in the source code repositories may be linear or as a directed acyclic graph depending on project methodology. For projects with multiple development branches, we have usually focused on the main development branch [51].

For defect prediction, source code repositories may be mined to extract static code, churn and collaboration metrics. Extraction of the static code metrics is dependent on the programming language. For static code metric extraction storage and data storage, we have built tools to avoid rework over the years [50], [52]. In the case of effort estimation, software repositories may be mined to extract the attributes related to the size of the software.

Issue management software are used to track the issues related to a software. It is the main repository of the quality assurance operations in a software project. We have used the issue management software to map the defects with the underlying source code modules and to analyze the issue handling process of organizations [53], [54].

Defect data may be used to label the defect prone modules for defect prediction or to assess software quality in terms of defect density or defect count. The ideal method to map defects to issues is associating source code changes with the issues. In some of our partners this strategy has been used efficiently. In this case, text mining is necessary to map issues with changes in the source code as accurately as possible. In our studies, we have frequently used the change set (commit) messages to identify the defect prone modules [49]. For the projects in which even the commit messages are not available, manual inspection of the changes with a team member may be attempted as a last resort for defect matching.

These defect data extraction methods can be used to use *within-company* defect data to train defect prediction models. In addition Turhan et al proposed certain methodologies to use *cross-company* data sources for cases where defect data is completely unavailable for individual software projects [55].

A few questions to answer when choosing the timespan for data extraction are as follows:

1. What is the specific software methodology employed by the organization for the given project?: Practical implementation of a particular methodology is always different from its theoretical definition.
2. Which releases and which projects should be used in the analytics project?: There may exist a difference among the data quality for different projects or releases. In addition, project characteristics such as size may change the outcome of experiments.
3. What is the estimated quality of the data?: It is usually beneficial to check the quality of the data with the quality assurance teams. Identifying problematic parts in the data early may help researchers to avoid rework.

Evaluation of qualitative results from surveys and questionnaire can also be done with quantitative methods. For metric extraction from qualitative data, digitizing the survey results to programmer friendly formats (e.g., plaintext, json, xml...) as early as possible would be useful.

Storage of metrics is a complicated task especially for datasets that may possibly be expanded in the future. These datasets may be used in future work with minimal effort if stored properly. The simplest method to store data is as a plain text file. For multi-dimensional data, we suggest the storage of the data as a light weight database such as Sqlite. Sqlite databases can be stored as a single file without the trouble of setting up a full database and since it is a single file the history of the database can be easily tracked. Every major language used frequently by software analysts such as R, Python and Matlab have native clients for Sqlite so it can be used. Sqlite can scale easily to store data with sizes of 1-10 GB in total [56].

Quality of the data, missing data and data sparsity are two common problems we have encountered frequently. We had to modify our data extraction methodology for each of these problems. Over the years, we have used several techniques to reduce the noise in our data and to address these problems. For example, in a project we had trouble in extracting certain set of metrics (cognitive bias metrics) from all developers. We have used matrix factorization to infer the missing values in our case [33]. One of our partners had several problems of keeping the defect data for their projects because of the lack of maturity in their processes. In their case, we had to train our model with cross-company data sources. Finding the right training data for their particular project had been a serious problem for us. For this goal, we evaluated sampling strategies to pick the right amount of data in our experiments [57].

Qualitative Data Extraction

Qualitative data could be extracted through questionnaires and surveys. Several guides from the social sciences literature can be studied for further reading on this subject [58].

Some input for the software analytics models cannot be extracted directly from the softtware repositories. We have extracted qualitative data for our models in many cases. For example, for modeling people we have used standardized tests to get cognitive biases of developers [29], [34]. We have also used questionnaires to assess process maturity in projects for reliability assessment models. We have used surveys to check the actual benefit of our analytics projects to software companies post-mortem too [49]. In our surveys, we found that even for successful projects that within-organization adoption of the analytics model may change over time without some vocal advocates.

It is costly to change qualitative data extraction methods after the initiation of data extraction. Therefore, the design of the qualitative data extraction methods is more time consuming than the quantitative data extraction methods.

Patterns in Data Extraction

Over the years, we have internally developed certain practices for extracting the relevant software data. Here is a short list of practices we recommend for the software analysts based on our experience:

1. Do not use proprietary data formats such as Microsoft Excel worksheets to store data since manual access to the data is time consuming. Although, these formats can be read through libraries, long term availability of these formats is not certain. In addition, binary formats make tracking the difference between versions very hard.

2. Track the version of the extracted data and the scripts.

3. Store your parameters for data extraction for each run. It is possible to forget a set of parameters that provided a particular outcome. Keeping the parameters stored in addition to the analytics output would be helpful to overcome this problem. For this goal separating the parameters from the source code would be helpful.

4. Look for possible factors that may introduce noise to the data. Internal and external factors might include noise into the data. You can control internal factors easily by checking your data model and scripts. On the other hand tracking the external factors is more tricky. Certain parts of the projects might be missing

5. Cross check possible problems in the extracted data with a quality assurance team member as early as possible.

6. All of the data extraction and analysis tasks and code should be executable with a single script. Redoing some operations may be impractical in some cases but if there is some problem in data extraction method, the researcher saves a lot of time using this practice. Remembering all the custom parameters and script names on every re-evaluation is time consuming.

1.4 Descriptive Analytics

The simplest way to get an insight of data is through descriptive statistics. Even a simple correlation analysis or visualization of multiple metrics may reveal hidden facts about the development process, development team or about the data collection process. For example, a graphical representation of commits extracted from a version control system would reveal the distribution of workload among developers, i.e., what percentage of developers actively develop on a daily basis [46], or it would highlight which components of a software system are frequently changed. On the other hand, a statistical test between metrics characterizing issues that are previously fixed and stored in an issue repository may identify the reasons for re-opened bugs [53] or reveal the issue workload among software developers [54]. Depending on the questions that are investigated, we can collect various types of metrics from software repositories; but it is inappropriate to use any statistical technique or visualization approach without considering the data characteristics (e.g. types and scales of variables, distributions). In this section, we present a sample of statistical techniques and visualizations that were used in our previous works and explain how they were selected.

1.4.1 Data visualisation

The easiest way to gain an understanding on the data collected from rich and complex software repositories is through visualization. Data visualization is a mature domain with significant amounts of literature on charts, tables and tools that could be used for visually displaying quantitative information [59]. In this section, we provide example use of basic charts such as box plots, scatter diagrams and line charts, in our previous analytics projects to better understand the software data.

Box plots are helpful for visualization as it allows to shape the distribution, central value (median), minimum and maximum values (quartiles) and gives clues about whether the data is skewed or contains noise, i.e., outliers. Figure 1.2 illustrates a box plot used to visualize the distribution of number of active issues owned by developers with different categories (FT, ST, Field) in a case study with a large scale software organization [54]. During this study, we used box plots to inform developers that FT category dominates the whole issues owned by developers, and it has a slightly higher median than ST and Field categories. Furthermore, outliers in each set (e.g. 48 FT issues are owned by a single developer) highlight potential noisy instances, or dominance of certain developers on the issue ownership.

Scatter diagram is another visualization technique to explore the relationship between two variables, i.e., two metrics that you want to monitor. During an exploratory study on Eclipse releases [60], we used scatter diagrams to observe the trend between number of edits on source files (first variable) and the number of days between edits (second variable) for three different file sets, i.e., files with beta-release bugs, other types of bugs and files with no bugs. Figure 1.4 shows the scatter diagrams for three file sets of an Eclipse release [60]. It is seen that that there is not a clear trend such as a positive monotonic relationship between the two variables. However, it is clear that files with beta-release bugs are concentrated on smaller regions; with very few edits done frequently (in small time periods). Based on these visualizations, we suggested that the developers may concentrate those files that are not edited too frequently in order to catch beta-release bugs [60].

Other types of charts are also useful for the purpose of visualizing software development, such as effort distribution of developers [46], issue ownership or developer collaboration [54]. For example, in a previous study, we identified the collaboration network of developers and what factors affected the team stability in a large, globally developed and commercial software [61]. We used line charts to visualize the developer collaboration based on the code that developers worked together. In another study, we also used line charts to depict the number of issues owned and being fixed by developers on a monthly basis (see Figure 1.3 from [54]). These charts are very easy to plot, and yet informative if they are monitored periodically, i.e., every week/ month or every sprint in agile practices, or per development team.

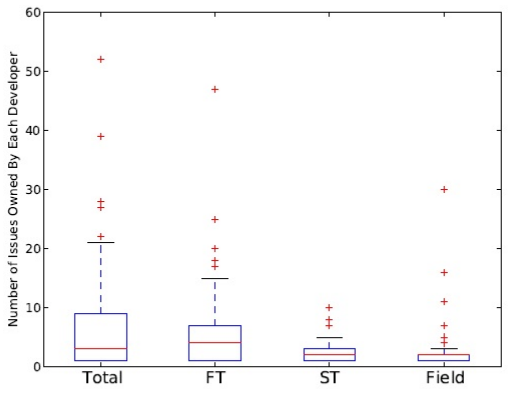

Figure 1.2: A box plot of issues owned by developers and their categories of a commercial software system [54]

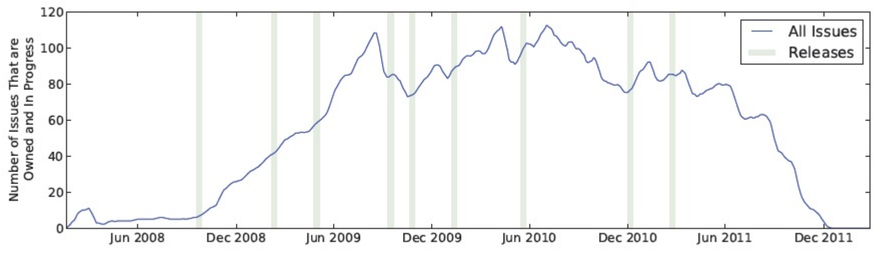

Figure 1.3: A line chart of activity in issue repository of Android system [54]

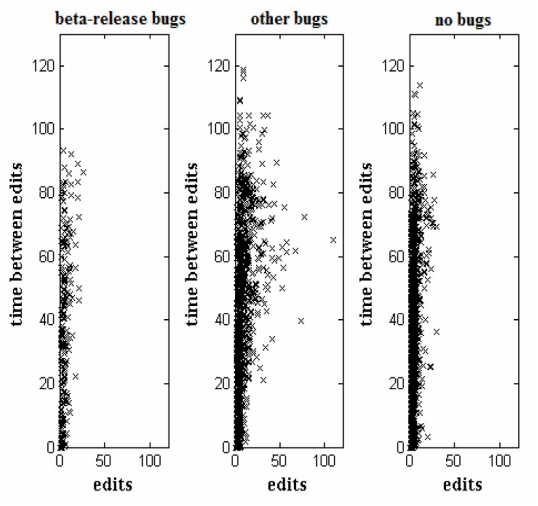

Figure 1.4: Three scatter diagrams between the number of edits and the time (in days) between edits depicting Eclipse release 2.1 [60].

1.4.2 Reporting via Statistics

Descriptive statistics such as minimum and maximum data points, mean, median or variance of software metrics are computed prior to building analytics, since these statistics are informative about the distributional characteristics of metrics, central values and variability. During an industry collaboration, it is a common practice to compute these descriptive statistics (e.g. in [49, 62, 54]) or use visualizations at the beginning of a software analytics project; whereas other techniques like statistical tests are more useful in next steps to reveal existing relationships between software metrics or to identify differences in terms of distributions of two or more software metrics.

Below, we summarize a set of statistical analysis techniques that were reported in our previous projects conducted with our industry partners. Detailed descriptions of these techniques and their computations can be found in fundamental books on statistics (see [63], [64]); we hereby explain the benefits of using these methods on software metrics to identify unique patterns.

Correlation analysis: This type of analysis identifies whether there are statistical relationships between variables and in turn, it helps to decide a set of independent variables that would be the input of a predictive analytics. Correlations can be computed among software metrics or between metrics and a dependent variable such as the number of production defects, defect category, the number of person-months (to indicate development effort), issue reopening likelihood, etc. Correlations between two variables can be measured using two popular approaches: Pearson product-moment correlation coefficient and Spearman's rank correlation coefficient.

Pearson correlation coefficient is a measure of the linear relationship between two variables, having a value between +1 (strong positive linear relation) and -1 (strong negative linear relation) [63]. The calculation of this coefficient is based on covariances of two variables and there are different guidelines for interpreting the size of the coefficient depending on the problem under study. For example, in empirical software engineering, a correlation coefficient above 75 per cent may be interpreted as a strong relation between the variables; whereas a correlation coefficient indicating a strong relation is quite small (37 per cent or more [65]) for empirical studies in psychology.

In a case study with a large software organization operating in telecommunications industry, we studied the linear relationship between confirmation bias metrics representing developers' thought processes and defect rates using Pearson correlation coefficient and found that there is a medium (21%) to strong (53%) relationship between the variables [34]. Based on correlation analysis, we subsequently filtered the metrics that are significantly correlated and used the resulting metric set to build a predictive analytics that estimates the number of defects of a software system.

Spearman's rank correlation coefficient is a non-parametric measure of statistical relationship between two variables [63]. Like Pearson correlation calculation, it takes values between +1 and -1 and is appropriate for both contiunous and categorical variables. Differently, Spearman's correlation calculation is based on the ranks that are computed from the raw values of the variables and is independent of the distributional characteristics of the two variables. Thus, the correlation can indicate any monotonic relationship in contrast to linear relationship in Pearson correlation.

In a prior study that investigates the experience, company size and reasoning skills of software developers and testers in four large to medium scale software organizations [32], we computed Spearman's rank correlation to observe the relationship between the number of bugs reported by testers and production defects, and between developers' reasoning skills and defect proneness of the code. Analyses show that there is a significant (rank coefficient: 0.82) positive relation between pre-release and production defects. We concluded that testers may report more bugs than the amount that developers fix before each release, and hence, as more bugs are reported, the number of production defects increase.

In another case study with a large scale software development organization, we used Spearman correlation to analyse the relationship between metrics that characterize reopened issues (issues closed and opened again during an issue life cycle) [53]. We found strong positive relationship between the lines of code changed to fix a reopened issue and the dependencies of a reopened issue to other issues, i.e., the higher the proximity of an issue to the others, the more lines of code is affected during its fix.

In summary, although these results may sometimes look trivial, they are very influential for software teams and development leads to observe the hidden facts about the development process in an organization, and/ or confirm assumptions that developers usually make in their daily decision making process. On the other hand, these types of correlation analyses only show whether there is a statistical relationship between two variables and the scale of that relationship. To further analyse what type of relationship (e.g. linear or non-linear) exists between these two variables, scatter diagrams can be used or other statistical tests (e.g. hypotheses testing) may be applied. Below, we provide examples of some of these tests that were used in our previous works.

Goodness of fit tests: These tests can be used to compare a sample (distribution) against another distribution or another sample that is (assumed to be) generated from the same distribution. Similar to correlation computation, there are several tests measuring the goodness of fit and the appropriate one should be chosen depending on the number of instances in a sample, the number of datasets that will be compared and distributional characteristics of the sample. For example, in a comparative study between test-driven development (TDD) and test-last programming, we used Kolmogorov-Smirnov test to compare the code complexity and design metrics of a software system that is developed using test-driven development (TDD) against a normal distribution [66]. Kolmogorov-Smirnov test is a non-parametric test for checking the equality of two continuous probability distributions or comparing two samples [64]. Our results in [66] show that none of the metrics are normally distributed and in turn, we applied analytics that are appropriate for samples generated from any type of distribution. In other studies, we applied the same goodness of fit test to compare development activities between experienced and novice team members of a software organization [31] or between developers and project managers [30].

In some of our studies, software metrics took categorical values and hence, other goodness of fit tests such as Chi-square test were more appropriate to compare the categorical metric values from two or more groups of samples. For example, during a case study with a software organization [33], Chi-Square test is used to compare the distributions of confirmation bias metrics, which characterize developers' thought processes and reasoning skills, among developers, testers and project managers. Based on the test results, we suggested to use confirmation bias metric suite in deciding the assignment of roles and in forming a more balanced structure in the software organization.

Differences between populations: Though goodness of fit tests can be used to compare the differences between two or more populations, i.e., samples, a better approach is to form a null hypothesis (H_0) and use statistical hypothesis tests to check if the null hypothesis is supported. For example, if we aim to check whether developers write more tests using TDD compared to test-last programming, we need to collect two samples that include the number of tests collected from a development activity using TDD and test-last. It is important that the samples are independently drawn from different populations. Later, we can form a null hypothesis as follows: *The difference between the number of tests written by developers using TDD and test-last approach has a mean value of 0.*

Applying a test (e.g. T-test, Mann-Whitney U test) rejects the null hypothesis, meaning that the mean values are significantly different between two development practices; or it can not reject the hypothesis, meaning that the difference of means is not significant enough for making a statistical claim [64].

Hypothesis tests cannot actually prove or disprove anything, and hence, they also calculate the power of the significance, e.g. p-value: 0.05 tells us that there is a 95% chance of the means being significantly difference. As this p-value increases, it is more likely that the difference happens by chance.

We have often used hypothesis tests to compare two or more samples representing different software systems, development methodologies (e.g. Mann-Whitney U test, also called Wilcoxon rank-sum test, in [66]) and/or software teams (e.g. Mann-Whitney U test in [32], Kruskal-Wallis analysis of variance with more than two teams in [34]). Alternatively, these tests can be used to compare the performance of two predictive analytics in order to decide which one to choose, e.g. T-test in [42, 19].

In summary, hypothesis testing techniques could be carefully selected by considering the information about sample distributions (e.g. T-test for normally distributed samples, non-parametric tests such as Mann-Whitney U test for the others) during a descriptive analytics process. We also suggest that descriptive statistics, reporting via visualization or through statistical tests could guide researchers or data analysts in software organizations in cases where there is a rich, multivariate and often noisy data.

1.5 Predictive Analytics

1.5.1 A Predictive Model for All Conditions

In this section, we will present our experiences from multiple industry collaborated projects in terms of predictive analytics. Throughout the section we will review multiple algorithms from simpler alternatives to more complex ones. However, our main intention is not to repeat commonly used machine learning (ML) algorithms, since there are more than enough ML books for that purpose [67, 68, 69]. We rather intend to share the lessons learned in the course of applying predictive analytics for more than a decade to industry-academia collaborated projects.

Often times, we will see that knowing the learner (a.k.a. prediction algorithm) is only the part of the story and the practitioner needs to be flexible and creative in terms of how s/he uses and alters the algorithm depending on the problem at hand. So, before going any further, just to set the expectations right, let us briefly quote a conversation with an industry practitioner. Following the presentation of a predictive analytics model, a practitioner from a large software company directed the following question: “Would your predictive model work under all conditions?” Conditions meaning different data sets and problem types. The short answer to that questions is: “No.” There is no predictive model that would yield high performance measures under all the different conditions. Therefore, if you are looking for a silver-bullet predictive model, this section will disappoint you. On the other hand, we can talk about a certain approach to predictive models as well as some likely steps to be followed, which have been applied in real life to a number of industry projects by the authors of this chapter.

Terminology: A predictive model is a specific use of a learner that is -not necessarily, but- often times aided with pre and post-processing steps to improve predictive performance [22]. We will talk about improving the performance in Section 1.5.3. For the time being let us focus our attention on the learners, which are machine learning algorithms that learn the known instances and provide a prediction for an unknown instance. The known instances refer to instances for which we know the dependent variable information, e.g. the defect information of software components. These instances are also referred as the *"training set"*. The unknown instances are the ones for which we lack the dependent variable information, e.g. the software components that have just been released, hence that do not have defect information. These instances are referred as the *"test set"*. In the example of defective and non-defective software components, a predictive model is supposed to use the training set and learn the relationship between the metrics defining software components and the defects. Using the learned relationship, the predictive model is expected to provide accurate estimates for the test set, i.e. the newly released components.

Go From Simple to Complex: One misleading approach to predictive modeling is to bluntly use whichever learner we come across. It may be possible to get a decent estimation accuracy with a randomly selected learner, but it is unlikely for such a random learner to be used by practitioners. For example, in a software effort estimation project for an international bank, every month we would present the results of the experiments to the management as well as the software developers [6, 45]. The focus of these presentations was never merely the performance, but how and why the presented algorithm could achieve the presented performance. In other words, merely using a complex algorithm, without a high-level explanation of how and why it applies to the prediction problem at hand, it is unlikely to lead to adoption by product groups. Therefore, it is a good idea to start the investigation with an initial set of algorithms that are easy to apply and explain such as linear regression [29], logistic regression [21, 53] and k-nearest-neighbor (k-NN) [70, 71]. The application of such algorithms will also serve the purpose of providing a performance baseline, so that we can see how much value-added the more complex additions will bring. These algorithms are relatively easier to understand and in most of the industry projects that the authors participated, they proved to be quite performant [21, 53, 29, 70, 71]. On the other hand, simplicity just for the sake of choosing algorithms that non-technical audiences can easily understand is also misleading. The simplicity of the algorithm for that purpose should be minimally effective on the decision process. If the best performing algorithm turns out to be some complex ensemble of relatively more difficult-to-understand learners, then so be it. But to make the case for deciding on a complex alternative, we should start simple and make sure that there is value-added with the complexity.

Linear regression is a predictive model that assumes a linear relationship between the dependent and the independent variables [67]:

$$y = X\beta + \varepsilon \tag{1.1}$$

The independent variables (defined by X) are multiplied by coefficients (of β) and we want to set up the coefficients such that the error (ε) is minimized.

We have utilized linear regression in various industry settings for regression problems (the problems in which the dependent variable is a continuous value). The focus of a project in which we collaborated with a large telecommunication company was the effects of confirmation bias on the defect density (defect density measured in terms of the count of defects, which is a continuous value, hence a regression problem). We opted to use linear regression in this project as the confirmation bias metrics were observed to be linearly related with defect counts [29]. The observation of liner relationship can be checked with the R^2 value as well as with plotting metric value against the defect count. In the confirmation bias project we defined defect density for each developer group as the ratio of the total number of defected files created/updated by that group to the total number of files that group created/updated. So as to visualize the effect of confirmation bias on software defect density, we constructed a predictive model based on linear regression, where confirmation bias metrics as the predictor (independent) variables and defect density as the response variable. Our results showed that 42.4% of variability in defect density could be explained by our linear regression model ($R^2 = 0.4243$)

Note that in the telecommunication company project, we used linear regression model so as to predict a continuous value (defect count) [29]. However, not all the predictive problems that we face in real-life predictive analytics involve continuous variables. In the case of discrete dependent variables, we can make use of logistic regression [21, 53]. For example, assume that we only know whether a software module is defective or not (but not the exact defect count), we can define the discrete classes of *"defective"* and *"non-defective"*. Such a grouping would give us a two-class problem (instead of a continuous-variable prediction problem). In the cases of two-class (a.k.a. binary) classification problems ($y_i = [0, 1]$), logistic regression is a frequently used prediction algorithm. The authors have employed logistic regression in various defect prediction projects [53]. The general formula of logistic regression is:

$$Pr(y_i = 1) = logit^{-1}(X_i\beta) \tag{1.2}$$

, where $Pr(y_i = 1)$ denotes the probability of y_i belonging to class 1, X is the vector of independent variable values and the β is the coefficient vector. One benefit of this predictive method is that -given the logistic regression provides a high accuracy- one can use the corresponding coefficient values in order to see the importance of different input variables. An example application of this approach can be found in one of our projects [53], where we have used logistic regression to analyze the possible factors behind issues being reopened in software development. For this purpose we fit a logistic regression model on the collected issue data, but our main aim is to understand which issue factors are most important (through the use of coefficient values). Therefore, for the analysis of factors leading to re-opened issues, logistic regression was an appropriate choice.

Another simple, yet quite successful learner for classification problems is Naïve Bayes [70, 71, 23]. Particularly for software defect prediction studies, the authors have observed that Naïve Bayes proved to be better than some more complicated rule based learners (like decision trees) [70]. As the name of the learner suggests, the Naïve Bayes classifier is based on the Bayes theorem, which states that our next observation depends on how new evidence will affect old beliefs:

$$P(H|E) = \frac{P(H)}{P(E)} \prod_i P(E_i|H) \tag{1.3}$$

In Equation 1.3, given that we have old evidences E_i and a prior probability for a class H, it is possible to calculate its next (posterior) probability. For example, for a classifier trying to detect defective modules in a software, class H would represent the class of defective modules. Then the posterior probability of an instance being defective ($P(H|E)$) is the product of the fraction of defective instances $P(H)/P(E)$ with the probability of each observation $P(E_i|H)$.

The success of this learner for defect prediction tasks, has also paved the way for us to use it in conjunction with other learners. For example, we have employed k-nearest-neighbor (k-NN) learner as a cross-company filter for Naïve Bayes [70]. Before going into how we made use of k-NN learner as a filter, let us briefly explain how it works: k-NN finds the labeled most similar k instances to the test instance. The distance is calculated via a distance function (like Euclidean distance, Hamming distance etc.).

k-NN can also be used both for classification [71] as well as regression problems [72]. If it is a classification problem, usually the majority vote (i.e. the majority class of k nearest-neighbors) is given as the predicted value. If it is a regression problem, then the mean or median of the dependent variable value k nearest-neighbors is given. Our use of k-NN as a filter is in the defect prediction domain, i.e. a classification problem [70], questions whether organizations without data of their own (so called *within-data*) can use the data from other organizations (so called *cross-data*). In our initial experiments we use Naïve Bayes as the predictive method and compare the performance when an organization uses *within-data* vs. *cross-data*. The performance results show that the use of *cross-data* yields a poor performance. This is understandable as the context of another organization may differ considerably. The we use k-NN to filter instances from the *cross-data*, e.g. instead of using all the instances of *cross-data* in a Naïve Bayes classifier, we first find the closest *cross-instances* to the test instance (using k-NN). Filtering only the closest instances improves the performance of *cross-data* to be very close to that of the *within-data*. The ability to use *cross-data* has been quite important in a number of our projects, because our observation is that initially the *within-data* will be quite limited at best. In other words, *"a practitioner will have to start the analysis with the* *data at hand"*. In such cases, the ability to use *cross-data* has helped us provide initial predictions for the *within* test data [45].

Other relatively more complex learners are also frequently employed in software engineering research as predictive methods. Neural networks [42] and decision trees [43, 73] are examples to such learners that were used by the authors. We will not go deep into the mechanics of these learners, however, we will provide the general idea and possible inherent biases. Because, having an idea of how these algorithms work and their biases aid a practitioner to choose the right learner for datasets of different characteristics.

Neural network (NN) are known to be universal approximators [74], i.e. they can learn almost any function. The neural networks are defined to be a group of connected nodes, where the training instances are fed into the nodes that form the input layer (see Figure 1.5) and the information is transmitted to hidden layer nodes. During transmission, each connecting edge applies a weight to the number it receives from the previous node. Although there is one hidden layer in Figure 1.5, there may be multiple hidden layers to model more complex functions. Finally, the values are fed into the output layer, where a final estimate for the for each dependent variable is attained. Note that instances can be fed into a NN one by one, i.e. the model can update itself one instance at a time as the training instances become available. However, the problem with the NNs is that they are sensitive to small changes in the data set [75]. Also, overt-raining a NN will have a negative impact for the future test instances that we want to predict. The authors addressed such issues related to NNs by arranging the NNs in the form of an ensemble [42]. Application of our NN ensemble on software effort estimation data revealed that unlike the problems inherent in a single NN, having an ensemble of NNs provides more stable and improved accuracy values [42]. We will futher discuss ensembles of other learners in the next section as a way to improve the predictive power of learners.

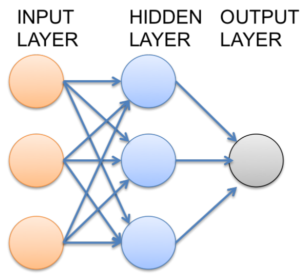

Figure 1.5: A sample NN with a single hidden layer

A capable and commonly used learner type is the decision trees [43, 73]. Unlike NNs that take instances one-by-one, decision trees require all the data to be available before they can start learning. Decision trees work by the recursively splitting data into smaller and smaller splits based on the values of independent features [76]. At each split the instances that are more coherent with respect to the selected features are grouped together into a *node*. Figure 1.6 presents a hypothetical decision tree that uses two features, $F1$ and $F2$. The way this decision tree would work for a test instance is as follows: If $F1$ value of this feature is smaller or equal to x, then decision tree would return the prediction of A. Otherwise, we will go down the node of $F1 > x$ and look at $F2$ value. If $F2 > y$ is true for our test instance, then we would predict B, otherwise we would predict C.

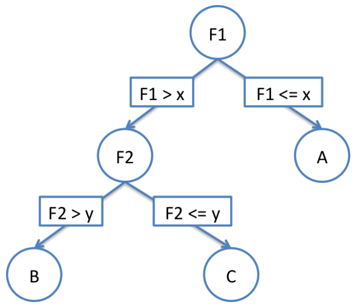

Figure 1.6: A sample decision tree that uses two features (F1 and F2).

Particularly for data sets where instances form interrelated clusters, it is a good idea to try decision tree learners as their assumptions overlap with the structure of the data. Software effort estimation data is a good example to this case, where instances (projects) form local groups that are composed of similar projects. Therefore, we made use of decision trees in different industry projects focusing on estimating the effort of software projects [45, 46, 43]. One of our different uses of decision tree learners on software effort data [43] includes the idea of grouping similar instances so as to convert the software effort estimation problem into a classification problem (recall that originally software effort estimation is a regression problem). By using decision trees we form software effort intervals, where an interval is identified by the training projects falling in the final node of a decision tree. Then the test instance is fed into the decision tree and it finds its final node, where the estimate is provided as an interval instead of a single value.

Things to keep in mind:

1. *Learner is only part of the picture:* Until now we have reviewed some of the learners that we successfully applied on different real-life software engineering data. We have also seen how these learners can be altered for different tasks, e.g. our use of k-NN as an instance filtering method [70] or the use of decision trees to convert a regression problem into a classification problem [43]. However, the application of the learner onto the data is only one part of the picture. There are possible pitfalls we identified over the course of different projects that a practitioner should be aware of when building a predictive model.

2. *Building a predictive model is iterative:* As you are applying different learners on the data you should consider including the domain experts (or customers) of the data in the loop. In other words, we recommend discussing the initial results and your concerns with the domain experts. The initial benefit of having the customer in the loop is that you can get early feedback about the quality of the data. For certain instances that your learner yields a poor performance, you can get a clear explanation as to what. Another benefit will be to identify potential data issues early on. If there is a feature that looks suspicious, the domain experts may give you a clear answer as to whether the data is wrong. For example, in one of the industry project, for some projects the bug count was surprisingly low, whereas the test cases for these projects were quite high. A domain expert who worked in this project may tell you whether this is indeed the case, or whether in this particular project bugs were not tracked consistently. Then you may decide not to include this unhealthy information into your model, as we did in that particular project.

3. *Automation is the key:* At any part of the project, you may require to re-run all your experiments. In an industry project, we learned halfway into the project that one of the assumptions we were told for a group of projects would be invalid for another group of projects developed within the last year. So we had to repeat all the analysis we made up to this point for the projects developed within the last year. In case we had not coded up our experiments, it would have taken a major time to repeat all the experiments. Therefore, it is beneficial to use machine learning toolkits like WEKA [69], but it is also critically important to have your experiments coded up for re-runs of the experiments. Another reason why it is important to code up your predictive models is customization. You may want to combine or alter different learners in a specific way, which may be unavailable through the UI of a toolkit. Coding up your experiments may provide you the flexibility of customization.

1.5.2 Performance Evaluation

Depending on the problem under study, performance evaluation of predictive analytics can be computed differently. Naturally, the output of a predictive analytics, i.e., prediction model, is compared against the data points in the test set, or against the actual data points after they are retrieved and stored at data repositories in software organizations. The type of output variables can be categorical or continuous, which determines the set of performance evaluation measures. For instance, in a defect prediction problem, the output of a predictive analytics can take a categorical value (e.g. defect proneness of a software code module, i.e., package, file, class, method, being defect-free (0) or defect-prone/ defective (1)) or it can take a continuous value ranging from zero to infinity (e.g. number of defect-prone code modules in a software product). For a model with a categorical output, a typical confusion matrix can be used for computing a set of performance measures. Table 1.5 presents a confusion matrix in which the rows represent the actual information about whether the software module is fixed due to a defect or not, whereas the columns represents the output of a predictive analytics.

All performance measures can be calculated based on four base measures presented in Table 1.5, namely True Positives (TPs), False Negatives (FNs), False Positives (FPs) and True Negatives (TNs). For example in the context of defect prediction, TPs are the actual number of defective modules that are correctly classified as defective by the model, FNs are the number of defective modules that are incorrectly marked as defect-free, FPs are the number of defect-free modules that are incorrectly classified as defective, and finally, TNs are the number of defect-free modules that are correctly marked as defect-free by the model. A predictive analytics ideally aims to classify all defective and defect-free modules accurately, i.e., FN = 0 and FP = 0. However, achieving the ideal case is very challenging in practice.

In our empirical studies, we have used six popular performance measures to assess the performance of a classifier: *Accuracy (ACC)*, *Recall (REC)* (also called *true positive rate* or *hit rate* or specifically in defect prediction, *probability of detection (PD) rate*), *False positive rate (FPR)* (also called *false alarms* in defect prediction), *Precision (PREC)*, *F-Measure* and *Balance (BAL)*. The calculations of these six measures can be seen below [21, 44, 22]:

$$ACC = (TP+TN)/(TP+TN+FN+FP) \tag{1.4}$$
$$REC = TP/(TP+FN) \tag{1.5}$$
$$PREC = TP/(TP+FP) \tag{1.6}$$
$$FPR = FP/(FP+TP) \tag{1.7}$$
$$F-Measure = 2(PREC*REC)/(PREC+REC) \tag{1.8}$$
$$BAL = 1-\sqrt{(1-REC)^2+(0-FPR)^2} \tag{1.9}$$

ACC, REC, PREC, F-Measure and BAL should be as close to 1 as possible, while FPR should be close to 0. However, there is an inverse relationship between REC and PREC, i.e., it is only possible to increase one (REC) with the cost of the other (low PREC). If these two are prioritized over all performance measures, it is useful to compute F-Measure which depicts the harmonic mean between REC and PREC.

Furthermore, there is a positive relationship between REC and FPR, e.g. an algorithm performs towards increasing REC with the cost of high false alarms. BAL is a measure that incorporates the relationship between REC and FPR; it calculates the distance between an ideal case (REC: 1, FPR: 0) and the performance of a predictive analytics.

Achieving high REC and PREC rates while lowering FPR, and in turn, achieving high BAL and F-Measure rates, has been a real challenge for us during building predictive analytics in software organizations. A good balance between these measures needed to be targeted according to business strategies in

Table 1.5: A typical confusion matrix used for performance assessment of prediction models

		Predicted	
		True	False
Actual	True	TP	FN
	False	FP	TN

these organizations. For example, software organizations operating in mission-critical domains (e.g. embedded systems), software teams aim to catch and fix as many defects as possible before deploying the production code. Thus, they prioritize achieving a high REC value over a low FPR from a predictive analytics. Conversely, software organizations operating in competitive domains (e.g. telecommunications), reducing the costs is the primary concern. Accordingly, they would like to reduce the additional costs caused by high FPR and achieve a balance between REC, PREC and FPR.

We have preferred to use REC, FPR and BAL in most of our prior work since reducing the cost of predictive analytics by lowering FPR has always been the focus of our industry partners. For example, in a case study with a software organization operating in telecommunications domain, we deployed a software analytics for predicting code defects [6]. Based on the user feedback, we calibrated the analytics to reduce FPRs while getting as high REC values as possible. The model that was later deployed in the company produced 87% recall and 26% false alarms [77]. In another study with a company operating in embedded systems, we built a predictive analytics for catching the majority of code defects with the cost of high false alarms [78]. We reported that an ensemble of classifiers was able to catch 82% recall and 35% false alarms. In other case studies, we have also studied different techniques for lowering false alarm rates in defect prediction research (e.g. algorithm threshold optimization in [79], increased information content of training data in [19, 17, 22], missing data imputation in [80]).

For localizing the source of software faults, we built a predictive analytics that extracts unique patterns stored in previous fault messages of a software application and assigns a newly arrived fault message to where it belongs [44]. In the context of fault localization, both REC and PREC are equally important since misclassification of faults has equal costs for both classes. In other words, localizing a fault to component A even though it belongs to component B can be as costly as localizing a fault to component B though it belongs to component A. Therefore, the proposed model in [44] managed to achieve 99% REC and 98% PREC in the first application, whereas it achieved 76% REC and 71% PREC rates in the second application.

Another predictive analytics was built for a software organization to estimate software classes that need to be refactored based on code complexity and dependency metrics [81]. The proposed model was able to predict 82% of classes that need refactoring (true positive rate) using 13% inspection effort by developers.

If predictive analytics produces a continuous output (e.g. total number of defects, project effort in terms of person-months), evaluating its performance requires either a transformation of the output to categorical values or use of other performance measures that are more appropriate for a continuous model output. For example, in a previous study, we built a software effort estimation model that would dynamically predict the interval of a project's effort, i.e., a categorical value that indicates which effort interval is the most suitable for a new project, rather than the number of person-months [73]. Since the output was categorical, performance measures that were used in defect prediction studies, i.e., REC, PREC, FPR and Accuracy, were used to assess the performance of the proposed effort estimation model.

A more convenient approach is to use other performance measures that are more suitable for evaluating the performance of a model that produces continuous output. Some of these measures that we have extensively used are *Magnitude of Relative Error (MRE)*, *Mean Magnitude of Relative Error (MMRE)* and *Predictions at level k (PRED(k)*. Calculation of each of these measures are listed below:

$$MRE_i = \frac{\mid x_i - \hat{x_i} \mid}{x_i} \tag{1.10}$$

$$MMRE = mean(allMRE_i) \tag{1.11}$$

$$PRED(k) = \frac{100}{N} \sum_{i=1}^{N} \begin{cases} 1 \text{ if } MRE_i \leq \frac{k}{100} \\ 0 \text{ otherwise} \end{cases} \tag{1.12}$$

Ideally, MRE and MMRE should be as low as 0, whereas Pred(k) should be close to 1. Instead of MMRE, *MdMRE (Median magnitude of relative error)* can also be used, since mean of a sample is always sensitive to outliers in that sample compared to *median.* In the context of software effort estimation, reducing MRE indicates that effort values are estimated with low error rates. As the error rate decreases, Pred rate increases. The value of k in Pred calculation is usually selected as (k= 25) or (k= 30), meaning that the percentage of estimations whose error rates are lower than 25% or 30%. Pred might be a better assessment criterion for software practitioners, since it shows the variation of the prediction error, i.e., what percentage of predictions achieve a level of error that was set by practitioners. Hence it implicitly presents both the error rate (k) and the variation of this error among all predictions made by the analytics.

We have used MMRE and Pred(25) measures during our empirical studies in the context of software effort estimation, and observed that there is not an optimal value for these measures that fit best for all software projects and teams. For example, in a prior study, one of our colleagues from our research laboratory built different predictive analytics to estimate the project effort in person-months [82]. Two datasets, i.e., one public dataset and another dataset collected from different software organizations, were used to train the predictive analytics. Results show that the best model on public dataset produced 29% MMRE, 19% MdMRE and 73% Pred(30), whereas it produced 49% MMRE, 28% MdMRE and 51% Pred(30) on the commercial dataset.

In other studies, we have proposed relatively more complex analytics to estimate software projects' effort in case of limited amount of training data. For example, the intelligence in our predictive analytics proposed in [42] works as follows: It has an associative memory that estimates and corrects the error of a prediction based on the algorithm's performances over past projects. Results using this associative memory show that we could achieve 40% MMRE, 29% MdMRE and 55% Pred(25) compared to a simple classifier that predicts with 434% MMRE, 205% MdMRE and 10% Pred(35) [42].

Based on the increase/ decrease in performance measures, the best performing algorithm can be selected during building predictive analytics. A raw comparison between the values of a performance measure may sometimes be problematic, i.e., the change in a measure might happen due to the sample used for training the algorithm and the improvements may not be statistically significant. So, even though an algorithm produces higher values in terms of a performance measure, say 30% over 27%, statistical tests should be used to confirm that the change does not happen by chance.

Statistical tests, as explained in Section 1.4.2, can be used to identify patterns of software data as well as to compare the performance of predictive analytics that are built with different algorithms (e.g. Mann-Whitney U test [66], Nemenyi's multiple comparison and Friedman test [45]). Box plots are also useful visualization techniques to depict the differences between the performance measures of two or more predictive analytics (e.g., [70, 22]). Other performance measures we used for predictive analytics that produce continuous output can be also found in [29].

Finally, we calculated context-specific performance evaluations, such as cost-benefit analysis (in terms of the amount of decrease (in LOC) in inspection effort for findings software defects [78, 77]) in empirical studies for software defect prediction. These types of analysis are also easy to understand and interpret for software practitioners as they represent the tangible benefits, i.e., effort reduction, of using predictive analytics.

1.5.3 Prescriptive Analytics

Once you decide on an algorithm, it is possible to improve the performance via some simple methods such as:

- Forming ensembles [78, 42, 83]
- Applying normalization [43, 73]
- Feature Selection through methods like Info-gain [70]
- Instance Selection via k-NN based sampling

Ensembles are one powerful way to combine multiple weak learners into a stronger learner [83]. The idea behind ensembles is that different learners have different biases (recall how NNs and decision trees are different from one another), hence they learn different parts of the data. When their predictions are combined they may complement one another and provide a better prediction. For example, in one of our studies, we employ a high number of prediction methods and combine them by simple ways such as taking the mean and median of the predictions coming from single learners [83]. However, before combining single learners, we pay attention to choose the successful ones, where we rank the learners and combine only the ones that perform a high performance. Such a selection of only the successful learners provided us ensembles that are consistently more successful than all the single learners. In other words, when forming an ensemble, it is better not to include the single learners whose performance is already low when they are run alone.

Normalizing the values of features is *"a must"* to improve learner performance if some of the features have very high values compared to others. For example, a feature keeping the number of classes (NC) defined in a project will have much smaller values compared to another feature that keeps the number of lines of code (LOC) in this project. If you are using a k-NN learner the impact of LOC will dominate over the impact of NC in a Euclidean distance calculation. A very common way we use for a number of different data sets is the min-max normalization [43, 73] , whose formula is given below, where X represents the feature vector and x_i represents a single value in this vector:

$$\frac{x_i - min(X)}{max(X) - min(X)} \tag{1.13}$$

Although a very high number of features may be available to practitioners to be included into a data set, it is usually the case that some of these features are less important than the others. We have observed in different scenarios that it is in fact possible to improve the performance of a learner by selecting a subset of all the features [70, 83]. There are multiple ways to select features such as info-gain [70], stepwise-regression [83] and linear discriminant analysis (LDA) just to name a few.

Similar to the idea of "not all features being helpful" for a prediction model, not all the instances are beneficial for a prediction problem too. There may be various reasons why certain instances should be filtered out of the data. For example, the instance may contain erroneous data. Or the data stored of the instance may be correct, but it may be so different from all the rest that it turns out to be an outlier. In either case, we may want to filter out such instances from the data set. We have discussed our use of k-NN based instance filters prior to a Naïve Bayes classifier [70], which is a good example of instance-based filtering. k-NN based instance-filters select only the closest training instances to the test instance, so that the learner uses only filtered out instances. Another algorithm that was inspired by the k-NN based filters is filtering by variance [72], where we have selected only the instances that form groups of low variance (of dependent variable value).

Table 1.6: A classification and application of some of the statistical methods for building a software analytics framework

Type of response variable (Output)	**Type of learning**	**Predictive analytics**		**Prescriptive analytics**
		Algorithms	**Performance measures**	
Categorical (e.g. defect proneness, issue reopening)	Classification	Logistic regression, Naïve Bayes, k-NN, Decision Trees	Base measures from confusion matrix (TP, TN, FP, FN). Other derived measures: Accuracy, Recall, Precision, False alarms, F-Measure, Balance	Normalization, feature and instance selection, ensembles
Continuous (e.g.number of defects, defect density, project effort in person months)	Regression	Linear regression, k-NN, Neural Nets, Decision Trees	MRE, MdMRE, Pred(k), R-squared	

The proposed methods of ensembles, feature and instance selection as well as feature normalization are a subset of possible ways to improve the performance of a learner. However, their application should not be interpreted as a must. Depending on the problem and data set at hand, a practitioner must experiment with their application separately or as combinations (e.g. feature and instance selection applied together). The combination that yields the best performance should be preferred. In summary, predictive analytics requires a clear definition of business challenges from a statistical point of view: from understanding data through visualizations and statistical tests to defining the input-output of a predictive analytics; from the selection of an algorithm to performance assessment criteria and finally, to improving the performance of the algorithm. Based on our previous experience, we provide Table 1.6 to guide software data analysts towards building a software analytics framework. Note that Table 1.6 is by no means an exhaustive list of all the possible algorithms for predictive purposes. However, it covers the selected algorithms that were employed in multiple industrial case studies that we discussed so far.

1.6 Road Ahead

In many real world problems, there are lots of random factors affecting the outcomes of a decision making process. It is usually impossible to consider all these factors and their possible interactions. Under such uncertainty, AI methods are helpful tools for making generalizations of past experiences in order to produce solutions for the previously unseen instances of the problem. These past experiences are extracted from available data, which represents the characteristics of the problem. Many data mining applications deal with large amounts of data and their challenge is to reduce this large search spaces. On the other hand, there exists domains with very limited amount of available data. In this case, the challenge becomes making generalizations from limited amounts of data.

In this context, software engineering is a domain with many random factors and relatively limited data. Nevertheless, in software domain, remarkably effective predictors for software products have been generated using data mining methods. The success of these models seems unlikely considering all the factors involved in software development. For example, organizations can work in different domains, have different process, and define/ measure defects and other aspects of their product and process in different ways. Furthermore, most organizations do not precisely define their processes, products, measurements, etc. Nevertheless, it is true that very simple models suffice for generating approximately correct predictions for software development time, the location of software defects.

One candidate explanation for the strange predictability in software development is that despite all the seemingly random factors influencing software construction the net result follows very tight statistical patterns. Building oracles to predict defects and/ or effort via data mining is also an inductive generalization over past experience. All data miners hit a performance ceiling effect when they cannot find additional information that better relates software metrics with defect occurrence, or effort intervals. What we observe from our past results is that the research paradigm, which relied on relatively straightforward application of machine learning tools, has reached its limits.

To overcome these limits, researchers use combinations of metric features from different artifacts of software, which we call information sources, in order to enrich the information content in the search space. However, these features from different sources come at a considerable collection cost and are not available in all cases. Another way to avoid these limits is to use domain knowledge.

So far in our research we have combined the most basic type of these features, i.e. source code measurements, with domain knowledge and we propose novel ways of increasing the information content using these information sources. Using domain knowledge, we show that, for example, data miners for defect prediction can easily be constructed with limited or no data.

Research in Artificial Intelligence (AI), programming languages, and software engineering share many common goals such as high level concepts, tools, and techniques: abstraction, modeling, etc. But there are also significant differences in the problem scope, nature of solution, and intended audience. The AI community is interested in finding solutions to problems; the software engineering community tries to find efficient solutions, and as a result it needs to tackle simpler, more focused problems.

When we define intelligence we have to be precise. What do we mean by intelligence? Which system is considered as intelligent? Those are the questions that are important to be able to create “smart” oracles. Building those systems becomes extremely important where we have large and distributed teams for software development. Users want an easy environment in which to switch between subsystems. For example in the defect prediction domain it is obvious that cooperative use of oracles with test benches is essential to support business decisions.

AI needs to be thought of as a large-scale engineering project. Researchers should build systems, and design approaches, that merge theory with empirical data, that merge science with large-scale engineering, that merge methods with expert knowledge, business rules and intuition.

In our previous work we had seen that static code attributes have limited information content. Descriptions of software modules only in terms of static code attributes can overlook some important aspects of software including: the type of application domain; the skill level of the individual programmers involved in system development; contractor development practices; the variation in measurement practices; and the validation of the measurements and instruments used to collect the data. For this reason we have started augmenting, and replacing static code measures with repository metrics such as past faults or changes to code or number of developers who have worked on the code, etc. In building oracles we have successfully modelled product attributes (static code metrics, repository metrics, etc.), and process attributes (organizational factors, experience of people, etc.). However, in software development projects people (developers, testers, analysts) are the most important pillar, but very difficult to model. It is inevitable that we should move to a model that considers Product, Process, and People (3Ps).

We believe that in defect and effort estimation, more value will come from better understanding of developer characteristics such as grasp of how social networks are formed and how they affect the defect-proneness and/ or effort allocation. Therefore research in this area will include input from other disciplines such as social science, cognitive science, economics and statistics.

Bibliography

[1] Kenett, R.S.: Implementing scrum using business process management and pattern analysis methodologies. Dynamic Relationships Management Journal pp. 29–48 (2013)

[2] Powner, D.A.: Software development: Effective practices and federal challenges in applying agile methods. Tech. rep., US Office of Public Affairs, GAO-12-681 (2012)

[3] Shearer, C.: The crisp-dm model: the new blueprint for data mining. Journal of Data Warehousing **5**, 13–22 (2000)

[4] Chrissis, M.B., Konrad, M., Shrum, S.: CMMI for Development: Guidelines for Process Integration and Product Improvement, 3rd edition edn. Addison Wesley (2011)

[5] Why is measurements of data management maturity important. Tech. rep., CMMI Institute (2014)

[6] Tosun, A., Bener, A.B., Turhan, B., Menzies, T.: Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry. Information and Software Technology **52**(11), 1242–1257 (2010)

[7] Misirli, A.T., Bener, A.: Bayesian networks for evidence-based decision-making in software engineering. IEEE Trans. Softw. Eng. **40**, 533–554 (2014)

[8] Menzies, T., Caglayan, B., He, Z., Kocaguneli, E., Krall, J., Peters, F., Turhan, B.: The promise repository of empirical software engineering data (2012). URL http://promisedata.googlecode.com

[9] McCabe, T.J.: A complexity measure. IEEE Transactions on Software Engineering **SE-2** (1976)

[10] Halstead, M.H.: Elements of Software Science, vol. 1. NewYork:Elsevier North-Holland (1977)

[11] Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Software Eng. **20**(6), 476–493 (1994)

[12] Fenton, N.E., Neil, M.: A critique of software defect prediction models. IEEE Trans. Software Eng. **25**(3), 1–15 (1999)

[13] Fenton, N.E., Ohlsson, N.: Quantitative analysis of faults and failures in a complex software system. IEEE Trans. Software Eng. **26**(8), 797–814 (2000)

[14] Oral, A.D., Bener, A.: Defect prediction for embedded software. In: 22nd International Symposium on Computer and Information Sciences (ISCIS 2007) (2007)

[15] Ceylan, E., Kutlubay, O., Bener, A.: Software defect identification using machine learning techniques. In: EURIMICRO SEAA 2006 (2006)

[16] Shull, F., Basili, V., Boehm, B., Brown, A.W., Costa, P., Lindvall, M., Port, D., Rus, I., Tesoriero, R., Zelkowitz, M.: What we have learnt about fighting defects. In: 8th International Software Metrics Symposium (2002)

[17] Turhan, B., Bener, A.: Weighted static code attributes for software defect prediction. In: 20th International Conference on Software Engineering and Knowledge Engineering (2008)

[18] Menziesi, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering **33**(1), 1–13 (2007)

[19] Turhan, B., Kocak, G., Bener, A.: Software defect prediction using call graph based ranking (cgbr) framework. In: 34th EUROMICRO Software Engineering and Advanced Applications (EUROMICRO-SEAA 2008) (2008)

[20] Kocak, G., Turhan, B., Bener, A.: Predicting defects in a large telecommunication system. In: ICSOFT, pp. 284–288 (2008)

[21] Tosun, A., Turhan, B., Bener, A.B.: Validation of network measures as indicators of defective modules in software systems. In: PROMISE, p. 5 (2009)

[22] Turhan, B., Misirli, A.T., Bener, A.: Empirical evaluation of the effects of mixed project data on learning defect predictors. Information and Software Technology **55**(6), 1101–1118 (2013)

[23] Misirli, A.T., Caglayan, B., Miranskyy, A.V., Bener, A., Ruffolo, N.: Different strokes for different folks: A case study on software metrics for different defect categories. In: Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics, pp. 45–51 (2011)

[24] Nagappan, N., Murphy, B., Basili, V.: The influence of organizational structure on software quality: An empirical case study. In: 30th International Conference on Software Engineering (ICSE 2008) (2008)

[25] Graves, T.L., Karr, A.F., Marron, J.S., Siy, H.: Predicting fault incidence using software change history. IEEE Trans. Software Eng. **26**(7), 653–661 (2000)

[26] Weyuker, E.J., Ostrand, T.J., Bell, R.M.: Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. Empirical Software Engineering **13**(5), 539–559 (2008)

[27] Mockus, A., Weiss, D.M.: Predicting risk of software changes. Bell Labs Technical Journal **5** (2000)

[28] Weyuker, E.J., Ostrand, T.J., Bell, R.M.: Using developer information as a factor for fault prediction. In: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, p. 8. IEEE Computer Society (2007)

[29] Calikli, G., Bener, A.B.: Preliminary analysis of the effects of confirmation bias on software defect density. In: ESEM (2010)

[30] Calikli, G., Bener, A.: Empirical analyses of the factors affecting confirmation bias and the effects of confirmation bias on software developer/ tester performance. In: PROMISE (2010)

[31] Calikli, G., Bener, A., Arslan, B.: An analysis of the effects of company culture, education and experience on confirmation bias levels of software developers and testers. In: ICSE (2010)

[32] Calikli, G., Arslan, B., Bener, A.: Confirmation bias in software development and testing: An analysis of the effects of company size, experience and reasoning skills. In: 22nd Annual Psychology of Programming Interest Group Workshop (2010)

[33] Calikli, G., Bener, A., Aytac, T., Bozcan, O.: Towards a metric suite proposal to quantify confirmation biases of developers. In: ESEM (2013)

[34] Calikli, G., Bener, A.: Influence of confirmation biases of developers on software quality: an empirical study. Software Quality Journal **21**, 377–416 (2013)

[35] Meneely, A., Williams, L., Snipes, W., Osborne, J.: Predicting failures with developer networks and social network analysis. In: 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE2008) (2008)

[36] Pinzger, M., Nagappan, N., Murphy, B.: Can developer-module networks predict failures? In: 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE 2008) (2008)

[37] Bird, C., Nagappan, N., Gall, H., Murphy, B., Devanbu, P.: Putting it all together: Using socio-technical networks to predict failures. In: Proceedings of the 2009 20th International Symposium on Software Reliability Engineering, ISSRE '09, pp. 109–119. IEEE Computer Society, Washington, DC, USA (2009)

[38] Zimmermann, T., Nagappan, N.: Predicting subsystem failures using dependency graph complexities. In: Proceedings of the18th IEEE International Symposium on Software Reliability, pp. 227–236. IEEE Computer Society (2007)

[39] Bicer, S., Caglayan, B., Bener, A.: Defect prediction using social network analysis on issue repositories. In: International Conference on Software Systems and Process (ICSSP 2011), pp. 63–71 (2011)

[40] Easley, D., Kleinberg, J.: Networks, Crowds, and Markets: Reasoning about a Highly Connected World. Cambridge University Press (2010)

[41] Kultur, Y., Turhan, B., Bener, A.: Enna: Software effort estimation using ensemble of neural networks with associative memory. In: 16th International Symposium on Foundations of Software Engineering (ACM SIGSOFT FSE 2008) (2008)

[42] Kultur, Y., Turhan, B., Bener, A.B.: Ensemble of neural networks with associative memory (enna) for estimating software development costs. Knowl.-Based Syst. **22**(6), 395–402 (2009)

[43] Bakir, A., Turhan, B., Bener, A.B.: A comparative study for estimating software development effort intervals. Software Quality Journal **19**(3), 537–552 (2011)

[44] Bakir, A., Kocaguneli, E., Tosun, A., Bener, A., Turhan, B.: Xiruxe: An intelligent fault tracking tool. In: International Conference on Artificial Intelligence and Pattern Recognition (2009)

[45] Kocaguneli, E., Tosun, A., Bener, A.B.: Ai-based models for software effort estimation. In: EUROMICRO-SEAA, pp. 323–326 (2010)

[46] Kocaguneli, E., Misirli, A.T., Bener, A., Caglayan, B.: Experience on developer participation and effort estimation. In: Euromicro SEAA Conference (2011)

[47] Lokan, C., Wright, T., Hill, P.R., Stringer, M.: Organizational benchmarking using isbsg data repository. IEEE Software **18**(5), 26–32 (2001)

[48] Bakir, A., Turhan, B., Bener, A.B.: A new perspective on data homogeneity in software cost estimation: a study in the embedded systems domain. Software Quality Journal **18**(3), 57–80 (2010)

[49] Misirli, A.T., Caglayan, B., Bener, A., Turhan, B.: A retrospective study of software analytics projects: In-depth interviews with practitioners. IEEE Software (2013)

[50] Caglayan, B., Misirli, A.T., Calikli, G., Bener, A., Aytac, T., Turhan, B.: Dione: An integrated measurement and defect prediction solution. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pp. 20:1–20:2. ACM, New York, NY, USA (2012)

[51] Caglayan, B., Tosun, A., Miranskyy, A.V., Bener, A.B., Ruffolo, N.: Usage of multiple prediction models based on defect categories. In: PROMISE, p. 8 (2010)

[52] Kocaguneli, E., Tosun, A., Bener, A.B., Turhan, B., Caglayan, B.: Prest: An intelligent software metrics extraction, analysis and defect prediction tool. In: SEKE, pp. 637–642 (2009)

[53] Caglayan, B., Misirli, A.T., Miranskyy, A.V., Turhan, B., Bener, A.: Factors characterizing reopened issues: a case study. In: PROMISE, pp. 1–10 (2012)

[54] Caglayan, B., Bener, A.: Issue ownership activity in two large software projects. In: 9th International Workshop on Software Quality, colocated with FSE (2012)

[55] Turhan, B., Menzies, T., Bener, A., Di Stefano, J.: On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering **14**(5), 540–578 (2009)

[56] Owens, M., Allen, G.: The definitive guide to SQLite, vol. 1. Springer (2006)

[57] Turhan, B., Kutlubay, F.O., Bener, A.B.: Evaluation of feature extraction methods on software cost estimation. In: ESEM, p. 497 (2007)

[58] Yin, R.K.: Case study research: Design and methods, vol. 5. sage (2009)

[59] Tufte, E.: The Visual Display of Quantitative Information, 2nd edition edn. Graphics Press (2001)

[60] Misirli, A.T., Murphy, B., Zimmermann, T., Bener, A.: An explanatory analysis on eclipse beta-release bugs through in-process metrics. In: 8th International workshop on software quality, pp. 26–33. ACM (2011)

[61] Caglayan, B., Bener, A., Miranskyy, A.V.: Emergence of developer teams in the collaboration network. In: CHASE workshop colocated with ICSE (2013)

[62] Kocak, S., Miranskyy, A.V., Alptekin, G., Bener, A., Cialini, E.: The impact of improving software functionality on environmental sustainability. In: ICT for Sustainability (2013)

[63] Hocking, R.R.: Methods and Applications of Linear Models: Regression and the Analysis of Variance, third edition edn. Wiley Series in Probability and Statistics (2013)

[64] Hollande, M., Wolfe, D.A., Chicken, E.: Nonparametric Statistical Methods, third edition edn. Wiley Series in Probability and Statistics (2014)

[65] Cohen, J.: Statistical power analysis for the behavioral sciences. Lawrence Erlbaum Associates Publishers., Hillsdale, New Jersey (1988)

[66] Turhan, B., Bener, A., Kuvaja, P., Oivo, M.: A quantitative comparison of test-first and test-last code in an industrial project. In: 11th International Conference on Agile Software Development (XP) (2010)

[67] Alpaydin, E.: Introduction to Machine Learning. MIT Press (2004)

[68] Bishop, C.M.: Pattern recognition and machine learning, vol. 1. springer New York (2006)

[69] Witten, I.H., Frank, E.: Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann (2005)

[70] Turhan, B., Menzies, T., Bener, A.B., Stefano, J.S.D.: On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering **14**(5), 540–578 (2009)

[71] Turhan, B., Koçak, G., Bener, A.B.: Data mining source code for locating software bugs: A case study in telecommunication industry. Expert Syst. Appl. **36**(6), 9986–9990 (2009)

[72] Kocaguneli, E., Menzies, T., Bener, A., Keung, J.W.: Exploiting the essential assumptions of analogy-based effort estimation. IEEE Trans. on Softw. Eng. **38**(2), 425–438 (2012)

[73] Bakir, A., Turhan, B., Bener, A.B.: Software effort estimation as a classification problem. In: ICSOFT (SE/MUSE/GSDCA), pp. 274–277 (2008)

[74] Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural networks **2**(5), 359–366 (1989)

[75] Geman, S., Bienenstock, E., Doursat, R.: Neural networks and the bias/variance dilemma. Neural computation **4**(1), 1–58 (1992)

[76] Breiman, L., Friedman, J., Stone, C.J., Olshen, R.A.: Classification and regression trees. CRC press (1984)

[77] Misirli, A.T., Bener, A.B., Kale, R.: Ai-based software defect predictors: Applications and benefits in a case study. AI Magazine **32**(2), 57–68 (2011)

[78] Tosun, A., Turhan, B., Bener, A.B.: Ensemble of software defect predictors: a case study. In: ESEM, pp. 318–320 (2008)

[79] Tosun, A., Bener, A.B.: Reducing false alarms in software defect prediction by decision threshold optimization. In: ESEM, pp. 477–480 (2009)

[80] Calikli, G., Bener, A.: An algorithmic approach to missing data problem in modeling human aspects in software development. In: Predictive Models in Software Engineering Conference (Promise) (2013)

[81] Kosker, Y., Turhan, B., Bener, A.: An expert system for determining candidate software classes for refactoring. Expert Systems with Applications **36**(6), 10,000–10,003 (2009)

[82] Baskeles, B., Turhan, B., Bener, A.: Software effort estimation using machine learning methods. In: 22nd International Symposium on Computer and Information Sciences (ISCIS), pp. 126–131 (2007)

[83] Kocaguneli, E., Menzies, T., Keung, J.W.: On the value of ensemble effort estimation. IEEE Trans. on Softw. Eng. **38**(6), 1403–1416 (2012)