Improving Reliability of Defect Prediction Models: from Temporal Reasoning and Machine Learning Perspective

DOCTORAL THESIS

for the Degree of a Doctor of Informatics

AT THE FACULTY OF ECONOMICS,

BUSINESS ADMINISTRATION AND INFORMATION TECHNOLOGY

OF THE

UNIVERSITY OF ZURICH

by

JAYALATH EKANAYAKE

from

SRI LANKA

Accepted on the recommendation of

PROF. DR. ABRAHAM BERNSTEIN, PH.D.

PROF. DR. HARALD C. GALL

2012

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, July 18, 2012

The Vice Dean of the Academic Program in Informatics: Prof. Dr. Harald C. Gall

Abstract

Software quality is an important factor since software systems are playing a key role in today's world. There are several perspectives within the field on software quality measurement. One such frequently used measurement (or metric) is the number of defects that could result in crashes, catastrophic failures, or security breaches encountered in the software. Testing the software for such defect is essential to enhance the quality. However, due to the rising complexity of software manual testing was becoming extremely time consuming task and consequently, many more automatic supporting tools have been developed. One such supporting tool is defect prediction models. A large number of defect prediction models can be found in the literature and most of them share a common procedure to develop the models. In general, the models' development procedure indirectly assumes that underlying data distribution of software systems is relatively stable over time. But, this assumption is not necessarily true and consequently, the reliability of those models is doubtful at some points in time.

In this thesis, therefore, we presented temporal or time-based reasoning techniques that improve the reliability of prediction models. By exploring four open source software (OSS) projects and one cost estimation dataset, we first disclosed that real-time based data sampling compared to random sampling improves the prediction quality. Also, the temporal features are more appropriate than static features for defect prediction. Furthermore, we found that the non-linear models are better than linear models for defect prediction. This implies, the relationship between project features and the defects is not linear. Further investigations showed that prediction quality varies significantly over time and hence, testing a model in one or few data samples is not sufficient to generalize the model. Specifically, we unveiled that the project features influence the model's prediction quality and therefore, the model's prediction quality itself can be predicted. Finally, we turned these insights into a tool that estimates the prediction quality of models in advance. This tool supports the developers to determine when to apply their models and when not.

Our presented temporal-reasoning techniques can be easily adapted to most of the existing prediction models for enhancing the reliability of those models. Generality, these techniques are easy-to-use, extensible, and show high degree of flexibility in terms of customization to real applications. More important, we provided a tool that supports the developers to make a decision about their prediction models in advance.

Zusammenfassung

Software Qualität ist ein wichtiger Faktor, da Software-Systeme eine Kernrolle in der heutigen Welt spielen. Es gibt verschiedene Perspektiven bezüglich der Messung von Softwarequalität. Eine häufig verwendete Metrik ist die Anzahl von Defekten, welche zu einem Absturz führen können, katastrophale Fehler oder Sicherheitsschwachstellen, welche in der Software entdeckt werden.

Das Testen der Software bezüglich solcher Defekte ist essenziell für die Qualitätssteigerung. Doch ist auf Grund steigender Softwarekomplexität das manuelle Testen eine äusserst aufwendige Aufgabe. Aus diesem Grund wurden automatisierte Unterstützungswerkzeuge entwickelt.

Ein solches Unterstützungstool sind Vorhersagemodelle für Defekte. Eine Grosszahl von Vorhersagemodelle für Defekte können in der Literatur gefunden werden. Die meisten verwenden einen allgemein üblichen Ansatz, um diese Modelle zu entwickeln. Dabei ist die grundlegende und indirekte Annahme, dass die zugrundeliegende Datenverteilung von Softwaresystemen relativ stabil über die Zeit ist. Doch ist diese Annahme nicht notwendigerweise korrekt. Aus diesem Grund ist die Verlässlichkeit solcher Vorhersagemodell für bestimmte Zeitpunkte zu bezweifeln.

Daher präsentierten wir in dieser Doktorarbeit temporale bzw. Zeit basierte Schlussfolgerungstechniken, welche die Verlässlichkeit von Vorhersagemodelle zu verbessern.

Basierend auf der Untersuchung von vier Open Source Software (OSS) Projekten, deckten wir auf, dass Echt-Zeit basierte Datenstichproben im Vergleich zu Zufallsstichproben die Vorhersagequalität verbessert. Des weiteren haben wir entdeckt, dass sich nicht-lineare Modelle besser für die Defektvorhersage eignen als lineare Modelle. Dies impliziert, dass die Beziehung zwischen Projekteigenschaften und Defekten nicht linear ist.

Weitere Untersuchungen ergaben, dass die Vorhersagequalität signifikant über die Zeit variiert. Deshalb ist Modellüberprüfung mittels einer oder mehrere Datenstichproben nicht genügend, um ein Modell zu verallgemeinern. Namentlich deckten wir auf, dass Projekteigenschaften einen Einfluss auf die Qualität des Vorhersagemodells haben und demzufolge die Qualität des Vorhersagemodelle selbst vorhergesagt werden kann.

Basierend auf diesen Erkenntnissen entwickelten wir schliesslich ein Werkzeug, welches die Vorhersagequalität von Modellen im Voraus bewertet. Dieses Werkzeug unterstützt die Entwickler bei der Entscheidung wann sie ihre Modelle anwenden sollen und wann nicht.

Unsere präsentierte temporale Schlussfolgerungstechnik kann einfach auf existierende Vorhersagemodelle angewendet werden, um deren Verlässlichkeit zu verbessern. Generell sind diese Techniken einfach zu verwenden, erweiterbar und zeigen einen hohen Grad an Flexibilität bezüglich ihrer Anpassbarkeit für echte Anwendungen. Doch bedeutender ist, dass wir ein Werkzeug erstellt haben, welche den Entscheidungsprozess der Entwicklern für die Verwendung ihrer Modelle unterstützt.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Prof. Dr. Abraham Bernstein for the continuous support of my PhD study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my PhD study.

Besides my advisor, my sincere thanks goes to my co-advisor Prof. Dr. Harald C. Gall for his encouragement, insightful comments, and hard questions.

Also, I would like to thank Prof. Dr. Burkhard Stiller of the Communications Systems Group for providing me necessary materials and guidance to prepare for the minor-subject.

I thank my fellow group mates in DDIS group: Jonas Tappolet, Philip Stutz, Floarea Serban, Dorothee Reinhard, Katharina Reinecke, Jonas Luel, Lorenz Fischer, Esther Kaufman, Peter Vorburger, Adrian Bachmann, Christoph Kiefer, Thomas Scharrenbach, and Amancio Bouza for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last four years.

I would like to express my sincere gratitude for my parents late Mrs. Lalatha Kumarihamy and Mr. U.W. Ekanayake, for giving birth to me at the first place and supporting me spiritually throughout my life.

I owe my loving thanks to my wife Upuli. She has lost a lot due to my research abroad. Without her encouragement and understanding it would have been impossible for me to finish this work.

Finally, I would gratefully acknowledge the financial support for my studies by the Sabaragamuwa University of Sri Lanka.

Table of Contents

List of Figures

List of Tables

Part I

Introduction, Background and Fundamentals

1

Introduction

`Software quality:` Software development organizations in the world are very keen on maintaining the quality of their products. The quality is so important since almost all of the institutions (airlines, banks, manufacturers, universities, etc.) that use software in their operations find themselves facing sharply increasing international competition. As our global society becomes more dependent on information (in contrast with capital or labor) in the production of goods and services, the pressures for higher quality, lower cost, and faster delivery for software products are increasing. Further we have more software developers eager to compete for customers. Due to these facts maintaining software quality becomes increasingly important for the developers. Software quality should emphasize three important factors: 1). Software requirements are the foundation from which quality is measured. Lack of conformity to requirement is lack of quality. 2). Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, it will result in lack of quality.
3). There is a set of implicit requirements that often go unmentioned (e.g. good maintainability). If software conforms to its explicit requirements but fails to meet implicit requirements, then the software quality is suspicious.

`How can we improve SW quality?` Quality requirements are increasingly becoming determining factors in selecting from design alternatives during software development. Therefore, it is important that the quality of software evaluated during the different stages of the development process. One way to determine the quality is predicting the quality in the early phases of the development. This prediction helps software developers to plan their resources according to the predicted quality and consequently, to improve the software quality.

`How can we predict the SQ?` During the past years, a large number of quality prediction models have been proposed in the literature. In general, the goal of these models is to predict the software quality using a set of directly measurable internal software metrics such as size and complexity metrics [Denaro et al., 2002],[Basili et al., 1996],[Subramanyam and Krishnan, 2003]. Some of the quality prediction models are based on metrics computed using data taken from the change and defect history of software projects [Khoshgoftaar et al., 1996],[Graves et al., 2000], [Nagappan and Ball, 2005a]. Further, researchers used metrics related to development process quality and testing process for defect prediction. However, these approaches are not widely used for quality prediction. Typically, models formulate relationships between the metrics and the quality

characteristics. After verifying these relationships, the models are fit for the quality prediction. It is typical of many linear models like regression based "data fitting" models and non-linear models like decision trees to a become common place for quality prediction in the literature [Ostrand et al., 2005], [Knab et al., 2006].

`Problems in quality prediction domain:` These prediction models reflect the programming style, the type of the software system, the application domain, and the profile of the company at the time the datasets are collected. In order to generalize these models, they should be tested against a considerable number of points in time and number of datasets drawn from diverse software projects. However, in the area of software engineering such data repositories are rare, due to two main reasons. First, there are not many companies that systematically collect information related to software quality (such as development effort, maintenance effort, reusability effort and bug reports). The second reason is that this type of information is considered confidential and the companies are not allowed to public the data. Due to the insufficient number of datasets, it is hard to generalize and reuse existing models. Since universal models do not exist, for a company, selecting an appropriate quality prediction model is a difficult, non-trivial decision. Given the fact that a quality prediction model is so crucial in software development and that it is difficult to select an appropriate universal model, the alternative is to improve the prediction quality of their local models. Now, the problem faced by software development organizations is: *how to improve the reliability of their models*.

The software development organizations rely on their quality prediction models and these models are highly localized to the company profiles. Therefore, companies are very keen on techniques that help to enhance their models' prediction quality. *The prime focus of this thesis is to fulfill this requirement by developing techniques that improve the prediction quality of the models. Further, these techniques are more generalized and can be incorporated to a range of prediction models*.

In the next section we explain the research problem and its relevance to the software community in detail. We first describe the term "software reliability", and the relationship of software reliability to software quality. Next, we discuss why the spotlight should focus on improving the prediction quality and then, briefly discuss time-dependent software evolution. To conclude this section we describe the importance of time-dependent prediction.

1.1 Motivation

The formal definition for software reliability is the probability of failure-free operation of a computer program for a specified time in a specified environment [Musa et al., 1987],[Lyu, 1996]. The Software reliability is the most important and most measurable aspect of software quality and it is very customer oriented. It is a measure of how well the program functions to meet its operational requirements. A failure is a departure of program operation from requirements. Failure intensity, an alternate way of expressing software reliability, is defined as failures occurring with respect to some time unit. An expression equivalent to the reliability figure given above is that a program

has a failure intensity of 0.025 failures for an hour of execution. A fault is a defect in a program that causes a failure. Fault or Defect-prediction in software systems is to discover segments or modules of software systems that lead to failures at the early stages of the development process. In this thesis we refer to quality prediction as predicting number or locations of residual defects and we use both these terms in the context.

1.1.1 Why are organizations interested in improving the prediction quality of models?

The software development paradigm has now reached maturity. Software products are becoming more and more complex. Eventually, this complexity may lead to several problems such as defects that cause failures, stability, maintainability etc. Further, manual evaluation of a complex software system is a time consuming and costly task. Hence, it has become important to develop tools that allow evaluating the software quality at the early stages of development. In general, predicting the quality of software is a complex task. Substantial amount of research effort has been spent to find methods for quality prediction over the last 30 years. As we mentioned above there are many approaches advocating models and metrics that purport to predict defects—as the fundamental quality factor. Generally, the past research has concentrated on the following three problem perspectives:

1. Predicting the number and the location of defects in the system;

2. Estimating the reliability of the system in terms of time to failure;

3. Understanding the impact of design and testing processes on defect counts and failure densities.

The defect prediction guides project managers to better decisions and as a result, they can quantitatively plan and steer the projects according to the expected number of bugs and their bug-fixing effort. Further, the defect prediction can also be helpful in a qualitative way whenever the defect location is predicted: testing efforts can then be accomplished with a focus on the predicted bug locations.

So far we understand the importance of defect prediction. In this regard, the reliability of the prediction is a crucial factor, since planning is done according to the prediction. We believe that powerful prediction models have been floated. But, little or almost no attention has been given for improving the prediction quality of already existing models. Therefore, organizations are curious about methods or techniques that improve the prediction quality of their models.

The subject matter of this thesis is to help organizations on how to improve software quality. Since this is the preliminary problem in software engineering domain we further narrow down

the problem as to how we can improve the defect prediction quality. We propose to include a time dimension into the feature space. We consider that time dimension has an enormous impact on prediction models, since events in software engineering happen chronologically and directly relate to the real-time. Furthermore, software systems' evolution may not consistent over time and the evolution rules may change over time. We explain about the evolution systems in details in the following section.

1.1.2 Time-Dependent Evolution

It is a well known fact that software systems have evolved over time. Hence, a software system can be equivalent to a dynamic system in the real-world. The dynamical system concept is a mathematical formalization for any fixed *rule* which describes the time dependence of a point's position in its ambient space. At any given time a dynamical system has a state given by a set of real numbers (a vector) which can be represented by a point in an appropriate state space (a geometrical manifold). Small changes in the state of the system correspond to small changes in the numbers. Analogous to the dynamical systems, the software has a state given by a set of parameters or metrics that describes the properties of the system at a given time point. Change in the state of the system results in the change in the metric values.

Tsymbal *et al.* [Tsymbal, 2004] found that the evolution rules of the dynamic system may not be a fixed rule that describes what future states follow from the current state due to the underlying concept change. They called this problem `concept drift`. Consequently, the models trained from the past metrics of the system are not any more fitting with the new metrics and regular updates for the models are necessary. Further, Widmer *et al.* [Widmer and Kubat, 1993] uncovered from daily experience that the meaning of many concepts heavily depends on implicit context. Changes in that context can cause radical changes in the concept. Further, Vorburger and Bernstein [Vorburger and Bernstein, 2006] said that context changes can be treated like concept shifts, since the underlying data generator (the concept) changes while moving from one context situation to another. These perceptions can be extended for software systems and for the models trained on the data collected from the software systems as well.

1.1.3 On the Importance of Time-Dependent Prediction

The most common practice in formulating prediction models is first, extract the metrics from snap-shot of a system at one time point and then use an algorithm to learn a relationship between the metrics and the expected number of defects. To evaluate this relationship, it is fed with the same set of metrics from another time point and the predicted values were compared with the observed ones facilitating an accuracy measure.

The common downside of these approaches is their temporally coarse evaluations. Usually, a bug prediction algorithm is evaluated, in terms of accuracy, in only one or a small number of points in time. Such an evaluation implicitly assumes that the evolution of a project and its

underlying data are relatively stable over time. The other common practice in formulating models is training models from data collected over a long period of time (history information). Also, in this method, they assumed that the evolution of the project is consistent over the observed time period. Formally, this assumption states that the evolution rule is fixed over time. However, according to the above section (Section 1.1.2), this assumption is not necessarily valid. Therefore, generalization of such models is difficult.

Generally, the developers of those prediction models have heavily neglected that software systems evolve over time, the evolution is not consistent and that the time factor is another dimension in the feature space. Due to this reason the prediction quality of those models is doubtful in some points in time. Time is an additional dimension to the feature space. It is neither a property of the system nor a variable implicitly included to the feature space. Thus the question remains as to how the time-dimension can be utilized to improve the prediction quality. The key objective of this thesis is to answer this question. To that end we first, formally present our problem in the next section.

1.2 Problem Definition

Up to this point we have explained most of the key aspects of the research problem along with an elucidation of the motivation for the research. The precise research question of this study is *How can prediction quality be improved using temporal reasoning techniques* and it is subdivided in to four solvable sub-problems.

Several researchers have developed defect prediction and cost estimation models based on historical information and some of them have included the time factor into the feature space – such as latest information – to build the models. However, there is very little or almost no research that formally investigated the impact of time-based or temporal data (see Section 6.1 for the definition of temporal and random sampling) on prediction quality. Therefore, this paves the way for our first Research Question:

Q1: Do time-based sampling techniques influence defect and cost prediction quality?

To find a formal answer for this question we define the following hypotheses.

H 1.1: *In the defect prediction domain, models trained on data collected from the temporal sampling technique are better than models trained on data collected from the random sampling technique.*

H 1.2: *In the cost estimation domain, models trained on data collected from the temporal sampling technique are better than the models trained on data collected from the random sampling technique.*

In the previous case we investigated significant of the time factor for prediction quality. However, it is still unclear that which kind of features – temporal or static features – improve the

prediction quality. Moreover, developers are often confused to select an appropriate learning algorithm to train models since there are several such algorithms – linear and non-linear – existing in the literature . In summery, these challenges create a problem of which features and algorithms make a reliable prediction model and this is raised in our second Research Question:

Q2: Which type of features – temporal or static – and models – linear or non-linear – are improving the prediction quality?

To empirically investigate this question we define following hypotheses (see Chapter 7 for the definition of temporal and static features).

H 2.1: *Models trained from temporal features are more predictive than models trained from static features.*

H 2.2: *Non-linear models are more precise on predicting defect than linear models.*

So far, a large number of prediction models have been developed [Catal and Diri, 2009] for predicting software quality. All they used a common procedure to train and test the prediction models. First, they constructed mostly static features and use those features to train prediction models. In order to evaluate these models, they computed the feature values from another time point and the predicted values are compared with observed ones. Usually, the evaluation is done with values computed in one or very few discreet time points. A typical question raised at this point is "whether such models are generalized?". This is an important problem to investigate and hence, it is our third Research Question:

Q3: Does the prediction quality remain constant in every time period?

To find a solution for the above question we define following hypotheses.

H 3.1: *Defect prediction quality varies over time.*

H 3.2: *There exist periods of stability and change in prediction quality.*

H 3.3: *There exist features that influence for prediction quality.*

In the previous experiment we hypothesized that prediction quality varies over time. If prediction quality varies over time then software developers should know particularly, when to rely on their prediction models and when not. The developers are very happy if there is a certain decision procedure that tells them about the quality of their models before applying them. However, now the problem is *when can we rely on bug prediction models?*. In order to find an answer to this question we define our forth Research Question:

Q4: Can the prediction quality of a model be estimated in advance?

We define the following hypothesis to answer the above research problem.

H 4.1: *A decision procedure can be defined to measure the quality of prediction models in advance.*

Summarizing, we state four research problems and define several hypotheses to address these research problems. All these problems are related to software quality and the solutions are helping to improve the software quality. We address each of these question separately in later chapters.

1.3 Our Approach

To address the mentioned issues related to defect prediction and in general to software quality, this thesis introduces time-based prediction techniques. Further, we use machine learning and data mining techniques. These techniques take the evolution aspects of software systems into consideration. Since there is no clear evidences to claim that the underlying concept of software systems constant over time, time-based prediction techniques help to improve the prediction quality.

1.4 Relevance to Software Engineering

In past years defect prediction has become an important task in Software Engineering. The relevance of defect prediction for Software Engineering lies in its ability to contribute improving the software quality by guiding the developer to better decisions. In the system engineering stage, it promotes quantitative specification of design goals, schedules, and resources required. It lets the developers determine the quality level during test and thus provides the means for evaluating the effect of various actions on quality so that it can be controlled. The defect prediction also helps in the better management of project resources.

It is important to understand that developers are still trying to improve the models' prediction quality due to the difficulty of finding universal model. Most of the models are data – and company profile – dependent and the prediction quality on new data is not guaranteed. Nonetheless, models' prediction quality is crucial for the success of software projects. This work provides easy-to-use methods by integrating temporal reasoning into the localized models.

Summarizing, the methods that are proposed to improve defect prediction quality substantially help software developers to improve the software quality.

1.5 Contribution to the Research Community

The core contribution of this thesis to the research community is the application and evaluation of temporal reasoning techniques that improve the defect prediction quality. Improving defect prediction quality improves software quality in general. Therefore, analyzing the above four research questions we support empirical software engineers to enhance software quality.

A few researchers used time-related software information such as last five years of software history to train models for predicting cost and defects of software projects. However, there is no any empirical investigation about the impact of time-related information on defect and cost prediction quality. Our investigation uncovers that temporal sampled data – exclusively prior information – is better than random sampled – prior and post information – data on defect prediction quality. However, there is no significant difference between these two sampling methods in terms when predicting cost. This is an interesting fact since the same sampling method has two different impacts on prediction models trained for two different tasks. Further, we are able to show that randomly sampled data without considering the time factor sometimes fails to spot important information and hence, negatively affects for the defect prediction quality.

A large number of features – temporal and static – can be extracted from software history information. Also, a large number of learning algorithms – linear and non-linear – have been developed so far. Selecting the most appropriate features and learning algorithms for training prediction models is a challenge for software engineers. We empirically show that the temporal features with non-linear models improve the defect prediction quality. Consequently, the future researchers can employ more temporal features rather static features as well as non-linear models for better prediction quality and hence, enhance the software quality. More importantly, we uncover that there is a non-linear relationship between the features and the defects.

Several models have been developed to predict defects in software systems. Those models were evaluated in one or very few points in time due to lack of data or other reasons and hence, the models were not generalized properly. Our experimental results show that keeping the target constant but varying the models is varying the prediction quality. Also, we show that keeping the model constant but varying the target is varying the prediction quality. This is a valuable information for the software developers since it conclude that not to rely on bug prediction models in every time. The reasons for such a high variability are more interesting and hence, we provide the influencing factors that signal for such higher variability in prediction quality. This information is very valuable since the developers can identify the incoming variability in defect prediction quality in advance. Further, we discuss the possible concept drift in software project which is a new phenomena for the software engineering community.

We found that the prediction quality varies over time. Therefore, we provide a tool to estimate

the prediction quality of a model in advance. In other words, this is a decision making procedure, which helps software developers to evaluate the model's prediction quality before hand. Basically, this tool proposes the developers when to use their prediction models and when not, which is a crucial information for the developers.

In conclusion, our approaches help to improve the defect prediction quality. We will show that our approaches are generally easy-to-use and can be applied to a range of prediction models. Further, these approaches help not only to improve the prediction quality but also to investigate hidden aspects of software projects such as variability in defect prediction quality, which is a new information and often valuable for the research community. Finally, we introduce temporal reasoning for the Software Engineering.

1.6 Outline of the Thesis

This thesis is structured in eight parts. The outline is as follows:

The introductory Part-I presents a summary of the most important related work that is relevant in the context of this thesis in Part-III-VI. Since this thesis associated with data mining and software engineering we start with relevant literature in machine learning field. Specifically, we review a number of relevant scientific efforts in the area of the software quality prediction in general and defect prediction in particular. As the main goal of this work is to improve the defect prediction quality, we revive the relevant approaches proposed in the literature. However, they are completely different than our techniques. The data quality issues are more important for empirical software engineering research and hence, we discus the relevant publications for improving the data quality. As we train models for cost estimation, we discuss some important models for cost prediction and the similarity between our models and those models. We discuss about the general issues in quality prediction models since we need to avoid those issues from our models. The chapter closes with brief presentation of concept drift since some of the observations of our experiments are signaling about possible concept drifts in software projects.

Part-II discusses the possible threats for all of our experiments.

In Part-III we briefly discuss the commonly used software engineering tools. Under this topic we brief about IDEs, VCS and BTS. More importantly, we discuss about the investigated software projects.

Part IV-VI contain the main contribution of this dissertation. We analyze the first research question – Do time-based sampling techniques influence prediction quality of defect and cost? – in Part-IV Chapter 6.1. To that end we conduct a comparison study between the temporal sampling

and random sampling techniques in different prediction domains – defect and cost – and that is the introduction point of the temporal reasoning techniques to the Software Engineering. In Part-IV Chapter 7 presents our second research equation. It investigates the importance of temporal features and non-linear models in defect prediction domain. In this chapter we empirically show that temporal features together with non-linear models improve defect prediction quality.

We showed that the prediction quality can be improved in Part-IV Chapter 7. However, now the question is "does the prediction quality constant in every time period?" (our third research question). We investigate this question in Part-V Chapter 8. We define three hypotheses to investigate this question and conclude that the prediction quality varies over time. Furthermore, we find the factors that influence for the prediction quality.

According to the conclusion from Part-V – prediction quality varies over time – another question arises – can we predict the prediction quality of a model?. In Part-VI Chapter 9, we investigate this question and we provide an approach to measure the model's prediction quality in advance. This is a valuable tool for the software developers since they can measure the prediction quality of their models before applying it and decide whether they rely on the prediction.

Part-VII summarizes and discusses the work presented in this thesis. Further, this part discusses the general limitations of the experiments, directions for future work and lists the bibliographic references. Finally, this part is closed with a short biographic note about the author of this dissertation.

2

Fundamentals and Related Work

The major contribution of this thesis is the introduction of temporal reasoning methods to improve the defect prediction quality. We present these methods in the remainder of this thesis. However, before getting into the details of those methods we revive the most important theories and related work in this chapter.

This thesis is associated with the research field **Data Mining** and **Software Engineering**. Therefore, we start with the theoretical foundation of learning algorithms that we apply in this thesis and data mining. Second, since our prediction models trained from software data we discuss about the similar prediction models in the literature as well as the prediction models for other purposes such as predicting refactoring, changes, cost, risk etc. We further review general issues of the prediction models and the steps taken to avoid those issues from our models. Finally, we review an interesting phenomena called *Concept drift*, which we believe existing in software projects, but not yet experimentally proven. We conclude this section with a short summary.

2.1 Machine Learning Algorithms

Machine learning is a scientific discipline that is concerned with the design and development of algorithms that allow computers to evolve behaviors based on empirical data, such as from software data, which we describe in the later section. A learner can take advantage of examples (data) to capture characteristics of interest of their unknown underlying probability distribution. Data can be seen as examples that illustrate relations between observed variables. A major focus of machine learning research is to automatically learn to recognize complex patterns and make intelligent decisions based on data; the difficulty lies in the fact that the set of all possible behaviors given all possible inputs is too large to be covered by the set of observed examples (training data). Hence the learner must generalize from the given examples, so as to be able to produce a useful output in new cases.

One branch of machine learning, *empirical learning*, is concerned with building models in the light of large number of exemplary cases, taking into account typical problems such as missing data and noise. Many such models involve classification and for these algorithms that generate decision trees are efficient, robust, and relatively simple [Breiman et al., 1984], [Quinlan, 1993].

Also, the neural networks, clustering algorithms, genetic algorithms (1950s), and support vector machines (1980s) involve in classification tasks in different practical problems. Other branch of machine learning, however, require the learned model to predict a numerical values associated with a case rather than the class to which the case belongs [Quinlan, 1992],[Wang and Witten, 1997].

In the Software Engineering domain many such learning algorithms play an important roll in various fields such as predicting defects, cost, changes etc. We revive about those models in the later sections. We also use such learning algorithms and the next subsection describes the theoretical foundation of these algorithms. Specially, we use inductive leaning models, which induce a general rule from a set of observed instances.

2.1.1 Statistical Learning Algorithms

In this thesis we train three machine learning algorithms to predict defects and the cost of software projects.

In most of our experiments we use class probability estimation (CPE) models. Specially, we use simple decision tree learner J48 [Witten and Frank, 2005] – a reimplementation of C4.5 [Quinlan, 1993][1] for generating a pruned or unpruned C4.5 decision tree – which predicts the probability distribution of a given instance over target classes. The J48 model is human readable and easy to be interpreted the generated rules. The J48 algorithm builds decision trees from a set of labeled training data using the concept of information entropy. It uses the fact that each attribute of the data can be used to make a decision by splitting the data into smaller subsets. J48 examines the normalized information gain (difference in entropy) that results from choosing an attribute for splitting the data. To make the decision, the attribute with the highest normalized information gain is used. Then the algorithm recurs on the smaller subsets. The splitting procedure stops if all instances in a subset belong to the same class. Then a leaf node is created in the decision tree telling to choose that class. But it can also happen that none of the features give any information gain. In this case J48 creates a decision node higher up in the tree using the expected value of the class. J48 can handle both continuous and discrete attributes, training data with missing attribute values and attributes with differing costs. Further it provides an option for pruning trees after creation.

Moreover, we use model trees M5P [Wang and Witten, 1997] – a reconstruction of Quinlan's M5 [Quinlan, 1992] algorithm – for our experiments. The model trees are just like ordinary decision trees, except that each leaf contains a linear regression model that predicts the class value of an instances that reach the leaf. Regression trees are a special case of model tree. According to the description in the book [Witten and Frank, 2005]:
regression and model trees are constructed by first using a decision tree induction algorithm to build an initial tree. However, whereas most decision tree algorithms choose the splitting attribute to maximize the information gain, it is appropriate for numeric prediction to instead minimize the intrasubset variation in

[1]The C4.5 is an extension of Quinlan's earlier ID3 (iterative dichotomiser 3 [Quinlan, 1986]) algorithm.

the class values down each branch. Once the basic tree has been formed, consideration is given to pruning the tree back from each leaf, just as with ordinary decision trees. The only difference between regression tree and model tree induction is that, for the latter, each node is replaced by a regression plane instead of a constant value. The attributes that serve to define that plane are generally those that participate in decisions in the subtree that will be prunedthat is, in nodes beneath the current one and perhaps those that occur on the path to the root node.

We use simple linear regression, which is an approach to modeling the relationship between a scalar variable y and one or more variables denoted X for our experiments. In linear regression, models of the unknown parameters are estimated from the data using linear functions. Such models are called linear models. The use of linear models for classification enjoyed a great deal of popularity in the 1960s; Nilsson [Nilsson, 1965] provides an excellent references.

2.1.2 Training (Learning) and (Validation) Testing of Models

In this paragraph we briefly describe the training and validation of models since these two terms are frequently appeared in this thesis. Further, we believe an additional context on these two tasks may help for the readers from Software Engineering background with little knowledge on data mining.

The term "learning" has several definitions from the philosophical point of view. But, we are not going to provide more details of these definitions in this context. However, the book [Russell and Norvig, 1995] provide more information on "learning" for the interesting readers. Generally, training or learning a model from data is a process that captures patterns automatically or semi-automatically in large quantities of data. There are several learning methods available. Among them we use inductive learning in which concepts are learned from sets of labeled examples or data – training set. We discuss about the structure of the datasets that we use for our experiments in the later sections.

Evaluating what's been learned is the key to making real progress. Testing or evaluating a model on training data is not a good indicator of performance on an independent test set since the information in the test set is already leaked into the model. Therefore, it is very important to use two data sets: one for training and the other for testing the model. When a vast supply of data is available, like software data, this is not an issue. In all of our experiments we use two separate datasets for training and testing the models.

The problem of measuring prediction performance based on limited data is an interesting and still controversial. There are many different techniques to counter this problem. Among those techniques, *cross validation* is gaining ascendance and is probably the evaluation method of choice in most practical limited-data situations. As we already mentioned, the decision tree models (J48) predict the class probability rather than the classes themselves, and others (M5P and linear models) involve in predicting numeric rather than nominal values. In most practical data mining problems the cost of misclassification errors is taking into consideration when it comes to the prediction quality of models. The cost depends on the error type – faults-positive(FP) or

faults-negative(FN) – and however, in our experiments, we consider both misclassification costs are equal.

In decision trees we measure the prediction quality using the Receiver Operating Characteristics (ROC) and Area Under ROC Curve (AUC) [Provost and Fawcett, 2001]. The acronym stands for ROC, a term used in signal detection to characterize the tradeoff between hit rate faults alarm rate over a noisy channel. The ROC curve indicates the performance of a classifier regardless of the initial class distribution of of the training data or error cost. This property of ROC curves is really useful for our experiments since our datasets are highly skewed (1:20 hasBugs vs. HasNoBugs). Further, Lessmann *et al.* [Lessmann et al., 2008] showed that AUC is a better performance measure of the prediction models. The ROC curves plot the number of positives (Sensitivity) included in the sample on the vertical axis (Y-axis), expressed as a percentage of the total number of positives, against the number of negatives (Specificity), on the horizontal axis (X-axis), expressed as the percentage of the total number of negatives. Figure 2.1 shows an example of ROC curve. Each point on the ROC plot represents a sensitivity/specificity pair corresponding to a particular decision threshold. A test with perfect discrimination (no overlap in the two distributions) has a ROC plot that passes through the upper left corner (100% sensitivity, 100% specificity). Therefore the closer the ROC plot is to the upper left corner, the higher the overall accuracy of the test. Further, the diagonal, which passes the lower left corner (0,0) and the upper right corner (1,1) represents the performance of random classifier. This implies the AUC=1 is perfect classifier and the AUC=0.5 is random.

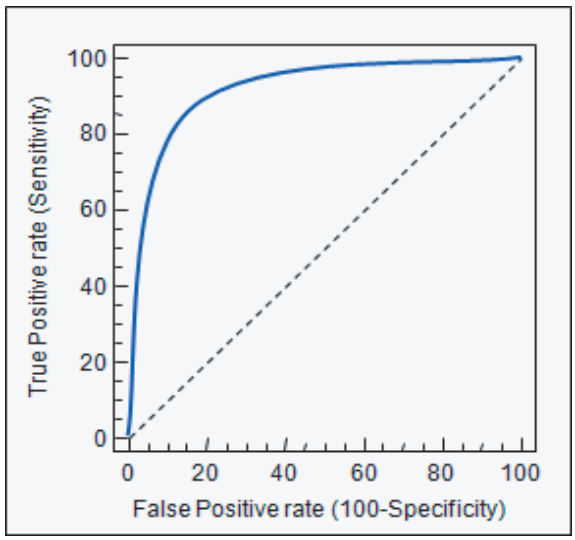


Figure 2.1: ROC curve [MedCalc Software,2011]

We measure the prediction quality of M5P and linear regression models using Pearson's and Spearman's rank correlation. Further, we use mean absolute error (MEA) and root mean square error (RMSE) as quality indicators of the models.

So far we introduced the learning algorithms that we use for our experiments as well as some

well known algorithms. Further, we discussed the prediction quality measures of those algorithms. In the next sections, we discuss the data that we used for training prediction models.

2.2 Software Data

The software history data is very crucial for empirical software engineering researches. Therefore, in this section we discuss about the software data storages, linking approaches between Version Control Systems (VCS) and Bug Tracking systems (BTS) and features or variables generation from software history.

2.2.1 Software Repositories

Software developers commonly use Concurrent Version System CVS[2] or SVN[3] for tracking the past changes (as VCSs) and bugzilla[4] or IssueZilla for recording the bug reports (as BTSs; see chapter 4 for more information). The information in these two databases is very valuable for empirical software engineering research. However, we have to link these two sources before using this data for research and that is not a trivial task. In the next section we discuss the existing approaches for linking these two data sources.

2.2.2 Linking CVS and Bugzilla

Information in the CVS and Bugzilla databases are frequently used for empirical software engineering research. However, linking these two data sources is challenge since it has a significant impact on data quality. Therefore, several researchers developed methods to counter this challenge.

Bachman *et al.* [Bachmann et al., 2010] presented a tool called `Linkster` that facilitates link reverse-engineering. They evaluated this tool, engaging a core developer of the Apache HTTP web server project to exhaustively annotate 493 commits that occurred during a six week period. Finally, they analyzed this comprehensive data set, showing that there are serious and consequential problems in the data. Further, Bachman and Bernstein [Bachmann and Bernstein, 2009] provided a software systems which hold such software data. First, they presented an approach for retrieving, processing and linking CVS and Bugzilla data. More importantly, they presented a step by step approach to retrieve, parse, convert and link these two data sources. Additionally, they introduced an improved approach for linking the CVS with the Bugzilla database.
Fischer *et al.* [Fischer et al., 2003b] developed a Release History Database (RHDB), which combines CVS and Buzilla and added missing data not covered by CVS such as merge points. Fisher

[2]http://www.nongnu.org/cvs/
[3]http://subversion.tigris.org/
[4]http://www.bugzilla.org/

et al. used a regular expression[5]. However, in this version no further verification of matching numbers has been done. This integration process of software data is new and we also use similar method to link CVS and Bugzilla databases. Later, they improved the links and built in verification mechanisms to overcome the wrong links issue and hence, improve the data quality [Fischer et al., 2003a].

Čubranić *et al.* [Čubranić and Murphy, 2003],[Čubranić et al., 2005] developed a tool called *Hipikat*, that integrates email discussion between the developers in addition to the CVS and Bugzilla data. They use small set of regular expressions[6] similar to Fischer *et al.* approach. Further, German [German, 2004] developed very similar tool called *softChange*. Compare to the above approach German used one regular expression[7]. Our expression for linking CVS and Bugzilla databases is similar to the one used by Čubranić *et al.* .

Śliwerski *et al.* [Śliwerski et al., 2005] adapted the technique presented by Fischer *et al.* and Čubranić *et al.* . They used two steps syntactic and semantic analysis to validate the potential links. In the syntactic level they used a regular expressions[8] in the CVS commit message as others did to identify the potential links to bug reports. Then, in the semantic level they used semantic analysis to validate the potential links. Specially, they used information about the bug report[9] and set the confidence level of semantic analysis to each potential link. Based on this confidence level they decided whether the potential link is valid or not.

Schröter *et al.* [Schröter et al., 2006] developed a dataset containing data from CVS and Bugzilla databases and the links between them. They first searched for potential references in the CVS commit massage for bug reports such as "Fixed 32555" or "Bug 4523". Such references are of low confidence at first. Therefore, they increased confidence level searching for more keywords such as "fixed", "bug" or pattern like "number followed by ". Further, they did not use bug report information to verify the identified links from the above methods. Zimmermann *et al.* [Zimmermann et al., 2007] used similar technique as Schrter *et al.* to link CVS and Bugizlla data sources.

We use CVS and Bugzilla data sources to construct datasets for training prediction models. We apply the approach described by Zimmermann *et al.* [Zimmermann et al., 2007] to link these two data sources. We cannot use their datasets since we need temporal based data for our experiments. However, this approach also contains few shortcomings. The manual verification of each bug report and the commit message is the only way of getting 100% confidence about the data (see [Bachmann and Bernstein, 2009]). But, this is really an expensive task. Nonetheless, we can safely say that our linking approach between CVS and Bugzilla is up to the standard compared to the similar studies in the literature.

[5] *e.g.*, "bugi?d?:?=?*?*(+)(.*)" or "b=(+)(.*)"

[6] *e.g.*, "Fix for bug 1234"

[7] "(#[0-9][0-9]+—bug?+#?[0-9][0-9]+)(,+#[0-9][0-9]+)*"

[8] *e.g.*, "bg[]*[0-9]+" or "show _bugċgiid=[0-9]+" and keywords "fix(e[ds]","bugs","defects")

[9] *e.g.*, the bug report has been resolved as FIXED at least once or the short description of the bug report is contained in the CVS log message

2.2.3 Features Extraction from Software History

Feature extraction is finding attributes (variables, features or metrics) that represent the problem of interest in the most appropriate way. A fundamental requirement for a prediction model is accurate determination of target variables from the features or attributes that represent the problem. We define several attributes (process attributes[10]) in the defect prediction domain to represent the problems. For detail description of features that we use for experiments can be found in Section 5.

2.2.4 Feature Generation

Generating set of features that reflect the underlying facts appropriately is not a trivial task. We extensively searched for features from the literature and selected features that are often used in the past researches. Further, we generated features using our own background knowledge. New features can be constructed by transforming or combining the original attributes. This approach is known as feature construction. It is often done by incorporating experts background knowledge about the problem domain.
Other methods for feature generation is without background knowledge. It applies optimized mathematical representation of the input attributes. The Principal Component Analysis (PCA) [Jolliffe, 2002] is such a method for finding the most descriptive features. The PCA finds the optimal linear axes transformation (rotation and stretching) by solving an Eigenvalue problem on the input attributes. However, we did not use PCA to find features because some vital features could be lost.

2.2.5 Feature Selection

Selecting the most suitable set of attributes that represent a problem, from a large set of attributes is also a challenging task. Some attributes might be irrelevant, redundant, or containing useful information only when combined together. We must select the best possible features before feeding them into the algorithm since this influence the quality of the prediction model as well as the computer resources (such as calculation time, memory usage *etc.*). The task of finding the best set of features is called `feature selection` [Blum and Langley, 1997], [Guyon and Elisseeff, 2003], [Hall and Holmes, 2003] and the next paragraphs discuss two feature selection methods that we use in this work.

Wrapper Methods
In machine learning a wrapper is an interpretative function that evaluates an expression to be

[10] process attributes are extracted for software history

tested and returns a value. This value allows to select the best alternative expression. In feature selection, the wrapper is the model evaluation based on different feature combinations. The evaluation result (e.g. the accuracy from a 10-fold cross validation) allows the identification of the best-performing model and thus, the best-performing feature combination. So, the best-performing feature combination is the feature combination to select from all features. There are three decisions to make to perform this kind of feature selection. First, what is the selection criterion to apply. Typically, the outcome of a classifier evaluation is the accuracy or the area under the ROC curve AUC [Provost and Fawcett, 2001]. These measures are the mostly used selection criteria following the rule: the higher, the better. Second, which algorithm to use. Although, the wrapper approach is concerned to be a black box approach to score the feature sub-sets, the algorithm choice has some influence on the results of the final model. Maybe the algorithm used by the wrapper has less discriminative power than the subsequent learner and thus, unintentionally, omits valuable information. Third, we have to determine the appropriate search strategy. Ideally, wrapper methods would make use of all possible feature combinations to determine the feature contributions (exhaustive, complete search). The state space of all possible feature combinations grows exponentially with the number of features. The number of states s grows as $s = 2^f$, where f is the number of total features. Therefore, the determination of the most relevant features using this kind of method is primarily a problem of computational complexity. As usual in computer science this problem can be represented as a search problem for which numerous solution strategies exist. Thus, most studies on wrapper methods are about finding the most efficient search strategy [Kohavi and John, 1997], [Opitz, 1999]. In feature selection, there are two fundamental search procedures, the forward and backward selection. Forward selection starts from scratch and adds new variables one-by-one while evaluating the optimal search path. The backward selection does the opposite: the search starts from a model based on all variables and eliminates one-by-one. The results of both approaches can differ due to non-independent variables and different stopping points when a certain quality threshold value is reached. In other wrapper application fields also other search techniques such as evolutionary search and simulated annealing are used.

Embedded Methods

Embedded methods perform variable selection during the training process of the definitive algorithm and are specific to given learning machines. In contrast to wrapper methods the embedded methods are not handling the algorithm as a black box. The decision trees (J48, M5P) have a built in mechanism to perform feature selection [Breiman et al., 1984]. More recent embedded methods guide their search for the feature sub-set by a fitness function which has to be optimized in order to reach maximal goodness of fit and minimal number of features [Cun et al., 1990],[Weston et al., 2003].

In training the prediction models we typically input different set of features and the feature selection is done by the models themselves using the above two methods. In Section 5 we provide a detailed description of each attribute that we generate from the software and cost estimation data.

2.2.6 Product Attributes

Product attributes or metrics are designed to capture the source code information and are are extracted for source codes. A number of different sets of product metrics have been suggested in past years. Among them McCabe [McCabe, 1976] introduced a set of metrics to measure the Cyclomatic complexity of a program. It directly measures the number of linearly independent paths through the program's source code. The Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program. Maurice Halstead [Halstead, 1977] introduced another complexity measure of a program as a means of determining a quantitative measure of complexity directly from the operators and operands in the module exclusively from source code. Chidamber and Kemerer metrics [Chidamber and Kemerer, 1994] developed a set of metrics to describe design structure of an object oriented (O-O) program. Several other metrics [Henry et al., 1981],[Tai, 1984],[Lorenz and Kidd, 1994] have been designed for capturing the source code structures. However, we omit discussing all of them.

More importantly we use process metrics rather than product metrics. Our decision to use only process metrics is further confirmed by Knab *et al.* [Knab et al., 2006]. They observed that process metrics are superior to product metrics in predicting defects.

2.3 Prediction Models in Software Engineering Domain

In the last 20 years, a large number of prediction models have been developed for various tasks in the Software Engineering such as bug, change, refactoring, cost, risk *etc.* prediction. Several research communities are actively engaged in developing models for the above tasks. Further, the novel research discussion forums such as PROMISE[11] or the Working Conference on Mining Software Repositories (MSR) [12] are very much focusing on those topics. Therefore, many publications are available on those topics.

2.3.1 Bug and Bug-Related Property Prediction Models

Predicting Location and Number of Bugs

A recent survey by Catal and Diri [Catal and Diri, 2009] discusses almost 100 publications. Hence, we remove most of the publications of defect prediction, but summarize few of them, which is more relevant to our study.

`Complexity Metrics (Product Matrix):`

In defect prediction domain, typically, metrics are defined to capture the complexity of software and builds models that relate these metrics to defects [Denaro et al., 2002]. Basili *et al.* [Basili et al., 1996] were among the first to validate that OO metrics are useful for predicting defect

[11] http://promisedata.org/
[12] http://2011.msrconf.org/

density. Subramanyam *et al.* [Subramanyam and Krishnan, 2003] presented a survey on eight more empirical studies, all showing that OO metrics are significantly associated with defects.

Post-release defects are the defects that actually matter for the end-users of a program. Only few studies addressed post-release defects so far: Binkley and Schach [Binkley and Schach, 1998] developed a coupling dependency metric and showed that it outperforms several other metrics; Ohlsson and Alberg [Ohlsson and Alberg, 1996] investigated a number of metrics to predict modules that fail during test or operation. Schröter *et al.* [Schröter et al., 2006] showed that design data such as import relationships also can predict post-release failures. Nagappan and Ball [Nagappan and Ball, 2005b] showed that relative code churn predicts software defect density; (absolute) code churn is the number of lines added or deleted between versions. Additionally, Nagappan *et al.* [Nagappan et al., 2006] carried out the largest study on commercial software so far: Within five Microsoft projects, they identified metrics that predict postrelease failures and reported how to systematically build predictors for post-release failures from history. Zimmermann *et al.* [Zimmermann et al., 2007] also used complex metrics to predict post-release bugs. They used Eclipse project data and uncovered that finding a single indicator or predictor for the number of defects is extremely unlikely and combination of complexity metrics can predict defects, suggesting that the more complex code it, the more defects it has.

Hassan [Hassan, 2009] proposed complexity metrics that are based on the code change process. They used concepts from information theory to define the change complexity metrics. They considered only feature introduction code change process in order to measure the change probability or entropy during specific time periods. The time periods were defined based on the calendar time from the start of the project, which is similar to the time frame definition in this thesis. The calculated entropy was then used to identify the subsystems or files with faults such that the subsystems or files that have higher entropy are at a risk of faults. Another similarity between our study and this study is the use of temporal data for training models.

Li *et al.* [Li et al., 2005] present a novel approach for the prediction of model parameters for software reliability growth models (SRGMs). These are time-based models using metrics-based modeling methods. They used three SRGMs, seven metrics-based prediction methods, and two different sets of predictors to forecast pre-release defect-occurrence rates. Our study also uses time-base prediction models to predict the location of defects. However, we predict defects in every possible time period which allows us a continuous analysis of the bug prediction quality.

`Product and Process Matrix:`
[Ostrand et al., 2005] and [Knab et al., 2006] both used code metrics and modification history to train regression models predicting the location and number of faults in software systems.

`Process Matrix:`
[Khoshgoftaar et al., 1996],[Graves et al., 2000], and [Nagappan and Ball, 2005a] all developed prediction models using software evolution data to predict future failures of the software systems. [Mockus and Votta, 2000a] showed that a textual description field of change history is essential to

predict the causes for this change. Further, they define three causes for a change: Adding new features, correcting faults and restructuring code to accommodate future changes (*i.e.* refactoring). we also use the change history for constructing features.

Kim *et al.* [Kim et al., 2007] analyzed the version history of 7 software systems to predict the most fault prone entities and files. The basic assumption is that faults do not occur in isolation, but rather in bursts of several related faults. Therefore, they cached locations that are likely to have faults: starting from the location of a known (fixed) fault, they cached the location itself, any locations changed together with the fault, recently added locations, and recently changed locations. By consulting the cache at the moment a fault is fixed, a developer can detect likely fault-prone locations. This is useful for prioritizing verification and validation resources on the most fault prone files or entities. In their evaluation of seven open source projects with more than 200,000 revisions, the cache selects 10% of the source code files; these files account for 73%-95% of faults—a significant advance beyond the state of the art.

All of the above methods for predicting defects share the following experimental procedure: first, they constructed several file-level and project-level features from the software history and use those features to train prediction models. Then, the feature values from another time period are computed and the predicted values are compared with observed ones. The common downside of these approaches is their temporally coarse evaluations. Usually, a bug prediction algorithm is evaluated, in terms of accuracy, in only one or a small number of points in time. This renders the generalization of models difficult, as such an evaluation implicitly assumes that the evolution of a project and its underlying data are relatively stable over time. However, this assumption is not necessarily valid [Tsymbal, 2004]. We will discuss about this issue in details in Sections 8.

In this study, we also use the software history to compute a set of features and some of our features are similar to these studies. In contrast, however, our feature set reflects almost all the changes to a file in past. In addition, we evaluate our prediction models throughout the project duration in order to show the variability in prediction quality over time and show the limited "temporal generalizability" of bug prediction models.

`Heuristics Based Bug Prediction:`
Hassan [Hassan and Holt, 2005] presented an approach (The Top Ten List) which highlights to managers the ten most susceptible subsystems (directories) to have a fault. Managers can focus testing resources to the subsystems suggested by the list. The list is updated dynamically as the development of the system progresses. They presented heuristics to create the Top Ten List and develop techniques to measure the performance of these heuristics. They applied their approach to six large open source projects (three operating systems: NetBSD, FreeBSD, OpenBSD; a window manager: KDE; an office productivity suite: KOffice; and a database management system: Postgres). Furthermore, they examined the benefits of increasing the size of the Top Ten list and study its performance. In our study we include few features to implement some of the heuristics in the Top Ten List such as recently fixed, recently changed, frequently fixed and frequently changed.

We partially base our work on the above mentioned related approaches by adopting some of their presented features.

Bug-fixing or Enhancement?

Antoniol *et al.* [Antoniol et al., 2008a] showed that there are many kind of issues such as defect fixing, enhancements, refactoring/restructuring activities and organizational can be found in the texts posted along with the issues in the bug tracking system. These different kinds of issues are simply labeled as bug for lack of a better classification support or of knowledge about the possible kinds. They investigated whether the text of the issues posted in bug tracking systems is enough to classify them into corrective maintenance and other kinds of activities and showed that alternating decision trees, naive Bayes classifiers, and logistic regression can be used to accurately distinguish bugs from other kinds of issues. Results from empirical studies performed on issues for Mozilla, Eclipse, and JBoss indicate that issues can be classified with between 77% and 82% of correct decisions.

However, Bachmann and Bernstein [Bachmann and Bernstein, 2009] revealed that a clear distinction between bug fixing and code enhancement activities would require manually verified datasets.

Causes for Bug Introduction

Sliwerski *et al.* [Śliwerski et al., 2005] introduced a refined approach to identify whether a change induced a bug fix. They combined a syntactic analysis, *i.e.* pattern matching, with semantic analysis. Semantic analysis compared the authors name of the CVS change with that of the developer responsible to propose bug fixing in Bugzilla. Consistency of dates and file versions were also part of their heuristics. They found that the larger a change, the more likely it is to induce a fix. They also found that in the Eclipse project, fixes are three times as likely to induce a later change than ordinary enhancements.

Purushothaman *et al.* [Purushothaman and Perry, 2005] presented an analysis of the software development process using change and defect history data. Specifically, they addressed the problem of small changes by focusing on the properties of the changes rather than the properties of the code itself. Their study reveals that 1) there is less than 4 percent probability that a one-line change will introduce a fault in the code, 2) nearly 10 % of all changes made during the maintenance of the software under consideration were one-line changes, 3) nearly 50 percent of the changes were small changes, 4) nearly 40 % of changes to fix faults resulted in further faults, 5) the phenomena of change differs for additions, deletions, and modifications as well as for the number of lines affected, and 6) deletions of up to 10 lines did not cause faults.

Baker and Eick proposed a similar concept of fix-on-fix changes [Baker and Eick, 1994]. Fix-on-fix changes are less general than fix-inducing changes because they require both changes to be fixes.

Mockus and Votta [Mockus and Votta, 2000b] hypothesized that a textual description field of a change is essential to understanding why that change was performed. Further, they claimed that

difficulty, size, and interval would vary strongly across different types of changes. They developed a program, which automatically classifies maintenance activity based on a textual description of changes and developer surveys showed that the automatic classification was in agreement with developer opinions. Tests of the classifier on a different product found that size and interval for different types of changes did not vary across two products. Further, they found strong relationships between the type and size of a change and the time required to carry it out. They also discovered a relatively large amount of perfective changes in the system they examined.

Aversano *et al.* [Aversano et al., 2007] presented a technique to identify bug-introducing changes to train a model that can be used to predict if a new change may introduces or not a bug. They represented software changes as elements of a n-dimensional vector space of terms coordinates extracted from source code snapshots. The evaluation of various learning algorithms on a set of open source projects looks very promising, in particular for KNN (K-Nearest Neighbor algorithm) where a significant tradeoff between precision and recall has been obtained.

Automatic Bug Assignment

Cubranic [Cubranic, 2004] used a text categorization approach with a Nave Bayes recommendation algorithm to assign bug reports to developers. They tested their approach on a collection of 15,859 bug reports from a large open-source project. The evaluation showed that their model, using supervised Bayesian learning, can correctly predict 30% of the report assignments to developers.

Canfora and Cerulo [Canfora and Cerulo, 2005] outlined an approach based on information retrieval in which they reported recall levels of around 20% for Mozilla. This work presents an approach that achieves a higher level of precision for these two projects.

Anvik [Anvik, 2006] showed that the task of triage can be eased by using a semi-automated approach to assign bug reports to developers. His work expands the previous works by Davor *et al.* and Canfora and Cerulo with more thorough preparation of data, the use of additional information beyond the bug description, the exploration of more algorithms, and the determination of a better performing algorithm. With his approach, he has reached precision levels of 57% and 64% on the Eclipse and Firefox development projects respectively.

Bug Property Prediction

There are several models that predict properties of bugs such as severity, lifetime, security *etc.*

`Severity of a Bug:`

Assigning a right severity is very important in deciding how soon it needs to be fixed. Though, clear guidelines exist on how to assign the severity of a bug, it remains an inherent manual process left to the person reporting the bug. To that end Menzies *et al.* [Menzies and Marcus, 2008] presented a new and automated method named SEVERIS (severity issue assessment), which assists the test engineer in assigning severity levels to defect reports. SEVERIS is based on standard text mining and machine learning techniques applied to existing sets of defect reports. They car-

ried out a case study on using SEVERIS with data from NASApsilas Project and Issue Tracking System (PITS) and the study results indicate that SEVERIS is a good predictor for issue severity levels, while it is easy to use and efficient.

Further, Lamkanfi *et al.* [Lamkanfi et al., 2010] investigated whether they can accurately predict the severity of a reported bug by analyzing its textual description using text mining algorithms. Based on three cases drawn from the open-source community (Mozilla, Eclipse and GNOME), they concluded that given a training set of sufficient size (approximately 500 reports per severity), it is possible to predict the severity with a reasonable accuracy (both precision and recall vary between 0.65-0.75 with Mozilla and Eclipse; 0.70-0.85 in the case of GNOME).

`Lifetime of a Bug:`
Song *et al.* [Song et al., 2006] used association rule mining to classify effort in intervals using NASAs SEL defect data. They found this technique to outperform other methods such as PART, C4.5 and Naive Bayes.

A self-organizing neural network approach for estimating effort to fix defects, using NASAs KC1 data set, was applied by Zeng and Rine [Zeng and Rine, 2004]. After clustering defects from a training set, they computed the probability distributions of effort from the clusters and compared it to individual defects from the test set to derive a prediction error.

Weiss *et al.* [Weiss et al., 2007] presented an approach to automatically predict the fixing effort of a bug. They used the Lucene framework to search for similar, earlier reports and use their average time as a prediction for a new bug report. They evaluated the approach using effort data from the JBoss project and their automatic predictions are close to the actual effort; for issues that are bugs, they are off by only one hour, beating naive predictions by a factor of four.

`Security of a Bug:`
Li *et al.* [Li et al., 2006] has shown that natural-language information can be used to classify root causes of reported Security Bug Reports for Mozilla and Apache HTTP Server. They used a natural-language model to identify the root causes of the security bugs. Based on their results, they determined that semantic security bugs (*e.g.,* missing features, missing cases) comprised 71.9-83.9% of the security bugs. These data provide guidance on what types of tools and techniques that security engineers should use to address most of their security bugs.

Podgurski *et al.* [Podgurski et al., 2003] use a clustering approach for classifying bug reports to prioritize and identify the root causes of bugs.

Geick *et al.* [Gegick et al., 2010] developed an approach that applies text mining on natural-language descriptions of bug reports to train a statistical model on already manually-labeled bug reports to identify security bug reports that are manually-mislabeled as non-security bugs. They evaluated the model's predictions on a large Cisco software system with over ten million source lines of code. Among a sample of bug reports that Cisco bug reporters manually labeled as non-security bug reports, their model successfully classified (78%) of the security bug reports as verified by Cisco security engineers, and predicted their classification as security bug reports with a

probability of at least 0.98.

The above mentioned bug classification models are not directly related to out study. But, we use some of the feature or attributes that they used for their models. Further, our prediction quality improving techniques can be adopted for those models. Next, we discuss the prediction models in other domain such as cost prediction, refactoring *etc.* in the following sections.

2.3.2 Cost Estimation Prediction

Cost estimation of a software project is another active topic in the Software Engineering domain. Several research publications can be found in this domain and number of models have been developed for a number of purposes:

1. Budgeting: the primary but not the only important use. Accuracy of the overall estimate is the most desired capability.

2. Tradeoff and risk analysis: an important additional capability is to illuminate the cost and schedule sensitivities of software project decisions (scoping, staffing, tools, reuse, etc.).

3. Project planning and control: an important additional capability is to provide cost and schedule breakdowns by component, stage and activity.

4. Software improvement investment analysis: an important additional capability is to estimate the costs as well as the benefits of such strategies as tools, reuse, and process maturity.

In this section we summarize few leading techniques.

Significant research on software cost modeling began with the extensive 1965 SDC study of the 104 attributes of 169 software projects [Nelson, 1966]. This led to some useful partial models in the late 1960s and early 1970s. The late 1970s produced a flowering of more robust models such as SLIM [Putnam and Myers, 1991], Checkpoint [Jones, 1991], SEER [Jensen 1983], and COCOMO [Boehm, 1981]. Although most of these researchers started working on developing models of cost estimation at about the same time, they all faced the same dilemma: as software grew in size and importance it also grew in complexity, making it very difficult to accurately predict the cost of software development. This dynamic field of software estimation sustained the interests of these researchers who succeeded in setting the stepping-stones of software engineering cost models. All of the above mentioned cost prediction models fall under model-based cost estimation techniques. The most commonly used techniques for these models include classical multiple regression approaches. In this thesis, our aim is to show the temporal sampling is better than random sampling in cost estimation and we also use the regression models for this task.

Beyond regression, several papers [Boehm, 1981], [Khoshgoftaar et al., 1995] discussed the pros and cons of one software cost estimation technique versus another and presented analysis results. We also train cost prediction models using temporal and non-temporal data and compare the prediction quality. Our comparison study is between the two data preparation methods instead of cost estimation techniques.

Expertise-based techniques are useful in the absence of quantified, empirical data. They capture the knowledge and experience of practitioners seasoned within a domain of interest, providing estimates based upon a synthesis of the known outcomes of all the past projects to which the expert is privy or in which he or she participated. Two techniques – Delphi technique [Helmer, 1966] and the Work Breakdown Structure [Baird, 1989] – have been developed which capture expert judgments. The obvious drawback of the expertise-based technique is that an estimate is only as good as the experts opinion, and there is no way usually to test that opinion until it is too late to correct the damage if that opinion proves wrong. Years of experience do not necessarily translate into high levels of competency. Moreover, even the most highly competent of individuals may sometimes simply guess wrong.

Learning-oriented techniques also popular in building cost prediction models. To that end Shepperd and Schofield [Shepperd and Schofield, 1997] did a study comparing the use of analogy with prediction models based upon stepwise regression analysis for nine datasets (a total of 275 projects), yielding higher accuracies for estimation by analogy. They developed a five-step process for estimation by analogy. According to Gray and McDonell [Gray and MacDonell, 1997], neural networks is the most common software estimation model-building technique used as an alternative to mean least squares regression. These are estimation models that can be trained using historical data to produce ever better results by automatically adjusting their algorithmic parameter values to reduce the delta between known actuals and model predictions. Gray, *et al.* go on to describe the most common form of a neural network used in the context of software estimation, a backpropagation trained feed-forward network. However, in our comparison study we use simple linear regression models since our goal is to compare two data sampling techniques – temporal vs. random.

Dynamics-based techniques explicitly acknowledge that software project effort or cost factors change over the duration of the system development; that is, they are dynamic rather than static over time. This is a significant departure from the other techniques highlighted before, which tend to rely on static models and predictions based upon snapshots of a development situation at a particular moment in time. The most prominent dynamic techniques are based upon the system dynamics approach to modeling originated by Jay Forrester nearly forty years ago [Porter, 1962]. Our dataset for cost estimation – ISBSG Release-9 – does not contain the information about the cost factors during the development life cycle. It contains only the final cost for a project and the properties of that project. However, in this thesis, we also highlight that software systems are equivalent to dynamical systems and the researchers should take this factor into consideration when training and testing prediction models.

Summarizing, though the above mentioned related work in the cost prediction claims that the models such as neural networks outperform linear regression model in predicting cost we use simple linear regression models for our experiment. Our goal is not to predict the final cost for a project, but compare two data sampling techniques.

2.3.3 Refactoring Prediction

Software history data can be used to deal with refactorings as well. Numerical measures can be used before applying a refactoring, to measure the (internal or external) quality of software, or after the refactoring, to measure improvements of the quality.

Demeyer *et al.* [Demeyer et al., 2000] proposed a set of heuristics for detecting refactorings by applying lightweight, object-oriented metrics to successive versions of a software system. They validated this approach with three separate case studies of mature object-oriented software systems for which multiple versions are available. The case studies suggested that the heuristics support the reverse engineering process by focusing attention on the relevant parts of a software system.

Simon *et al.* [Simon et al., 2001] use distance-based cohesion metrics to detect where in a given piece of software needs for refactoring. Due to the fact that the software developer is the last authority they provided software visualization tool to support the developers judging their products. They demonstrated this approach for four typical refactorings and presented both the tool supporting the identification and case studies of its application.

Kataoka *et al.* [Kataoka et al., 2002] proposed a quantitative evaluation method to measure the maintainability enhancement effect of program refactoring. They focused on the coupling metrics to evaluate the refactoring effect. By comparing the coupling before and after the refactoring, they evaluated the degree of maintainability enhancement. They applied this method to a certain program and showed that the method was really effective to quantify the refactoring effect and helped developers to choose appropriate refactorings.

Ratzinger *et al.* [Ratzinger et al., 2007] found that attributes of software evolution data can be used to predict the need for refactoring in the following two months of development. They used information in the CVS as input into classification algorithms to create prediction models for future refactoring activities. Different state-of-the-art classifiers were investigated such as decision trees, logistic model trees, propositional rule learners, and nearest neighbor algorithms. They predicted the refactoring proneness of object-oriented systems with high precision and recall values. More importantly, in our work we also use two-months training window to train prediction models.

2.4 Improving Reliability of Prediction

Although a number of approaches have been developed for quality prediction for software, little work has been done to improve the prediction quality of existing models.

Bouktif *et al.* [Bouktif et al., 2006] proposed an approach for the combination and adaptation of software quality predictive models. Quality models are decomposed into sets of expertise. This approach can be seen as a search for a valuable set of expertise that when combined form a model with an optimal predictive accuracy. Since, in general, there will be several experts available and each expert will provide his expertise, the problem can be reformulated as an optimization and search problem in a large space of solutions. They presented how the general problem of combining quality experts, modeled as Bayesian classifiers, can be tackled via a simulated annealing algorithm customization. The general approach was applied to build an expert predicting object-oriented software stability, a facet of software quality. However, this approach suffers from computational complexity issue. Further, we also provide techniques for improving the reliability of prediction models, but our approach is based on temporal reasoning techniques with less complexity.

Bouktif *et al.* [Bouktif, 2004] suggested two general approaches to software quality prediction. They consisted of combining/adapting a set of existing models. The process is driven by the context of the target company. These approaches are applied to OO software stability prediction. Analogous to the above approach, this approach has also the same complexity issue. Further, the outcome of this method can be applied only for the target company.

2.5 General Issues in Prediction Models

Fenton *et al.* [Fenton and Neil, 1999] provided a critical review of defect prediction models. They claimed that there are a number of serious theoretical and practical problems in many studies. In particular, they mentioned six issues regarding defect prediction models:(1) unknown relationship between defects and failures, (2) problems with the multivariate statistical approach, (3) problems of using size and complexity metrics as sole predictors of defects, (4) problems in statistical methodology and data quality and (5) false claims about software decomposition. In this thesis, we tried to avoid the above mentioned issues much as possible. Nevertheless, this was not completely possible, and therefore, we mention those problems in Section 3. Additionally, to ensure methodical soundness, we employee the methods describing in Zimmermann *et al.* [Zimmermann et al., 2007] to link the CVS and the Bugzilla databases.

Lessmann *et al.* [Lessmann et al., 2008] proposed a framework to compare defect prediction models. The framework addressed three potential issues related to model comparison process; (1) comparing classifiers over one or a small number of proprietary data sets, (2) relying on accuracy indicators that are conceptually inappropriate for software defect prediction, and (3) cross-study comparisons and limited use of statistical testing procedures to secure empirical findings.

Further, they proposed that in general, metric-based classification reasonably better than other classification models and the AUC is a better performance indicator of prediction models. In our investigation, we also use metric-based classification models and more importantly, we use AUC as performance indicator of our models. Moreover, we use statistical tests where appropriate.

To our knowledge any study has not pointed out about the assumption – evolution of a project and its underlying data are relatively stable over time. As we mentioned all most all the models have been developed under this assumption. However, our experiment results suggest that it is worth to be paid attention for this issue in the future researches.

2.6 Software Engineering Data Quality

Software Engineering data is widely used for empirical software engineering researches, including in this thesis. Therefore, in this section we briefly review few most relevant related work about data quality.

Bettenburg *et al.* [Bettenburg et al., 2007] conducted a survey among Eclipse developers to determine the information in reports that they widely used and the problems frequently encountered. They found that steps to reproduce and stack traces are most sought after by developers, while inaccurate steps to reproduce and incomplete information pose the largest hurdles. Surprisingly, developers are indifferent to bug duplicates. Such insight is useful to design new bug tracking tools that guide reporters at providing more helpful information. They also presented a prototype of a quality-meter tool that measures the quality of bug reports by scanning its content.

Antoniol *et al.* [Antoniol et al., 2008b] pointed out the lack of integration between version archives and bug databases. Providing such an integration allows queries to locate the most faulty methods in a system. While the lack of integration was problematic a few years ago, things have changed in the meantime: the [Zimmermann et al., 2007], [Bachmann et al., 2010] derived new approaches to integrate those two databases and show that their approaches enhanced the link quality.

Ayari *et al.* [Ayari et al., 2007] attempted to shed some light on threats and difficulties faced when trying to integrate information extracted from Mozilla CVS and bug repositories. In the reported Mozilla case study, they observed that available integration heuristics are unable to recover thousands of traceability links. Furthermore, Bugzilla classification mechanisms do not enforce a distinction between different kinds of maintenance activities. They used linking approaches published by [Fischer et al., 2003a] and [Śliwerski et al., 2005]. So, we use similar linking approach developed by [Fischer et al., 2003b], which is the previous version of [Fischer et al., 2003a].

Liebchen and Shepperd [Liebchen and Shepperd, 2008] assessed the extent and types of techniques used to manage quality within software engineering data sets. They performed a systematic review of available empirical software engineering studies and found only 23 out of the many hundreds of studies assessed, explicitly considered data quality.

Summarizing, the data quality is very important issue when it comes to empirical research

in the Software Engineering. We very much concerned on this issue when collecting software data. Specially, we apply very similar approach to the one described by Zimmermann [Zimmermann et al., 2007] to link the CVS and Bugzilla databases. Further, the Zimmermann's approach overlaps with some other well known approaches such as [Fischer et al., 2003b], [Śliwerski et al., 2005] *etc.* and hence, the results of our experiments will not significantly change on the linking approach. we can guaranty our data quality.

2.7 Concept Drift

This thesis does not investigate the concept drift, which is a notion from machine learning that refers to change in the data generation process. However, from observations of our experiments we repeatedly ask ourselves if the cause behind is the concept drift. Therefore, in this section we provide a few publications related to concept drift studies.

A concept is the underlying rule that generates the data set. In machine learning an algorithm usually learns a model, which should be as close as possible to the concept. When a concept changes (drifts) the algorithms model needs to change too. Alexey Tsymbal provides a survey on concept drift research [Tsymbal, 2004]. He defined the concept drift as follows:

In the real world concepts are often not stable but change with time. Typical examples of this are weather prediction rules and customers preferences. The underlying data distribution may change as well. Often these changes make the model built on old data inconsistent with the new data, and regular updating of the model is necessary. This problem, known as concept drift, complicates the task of learning a model from data and requires special approaches, different from commonly used techniques, which treat arriving instances as equally important contributors to the final concept.

Further, he claimed that models trained from past metrics of the system are not any more fitting with the new metrics and regular updates for the models are necessary.

Widmer *et al.* [Widmer and Kubat, 1993] uncovered from daily experience that the meaning of many concepts heavily depends on implicit context. Changes in that context can cause radical changes in the concept. These perceptions can be extended for software systems and the models trained on the data collected from the software systems too.

To our knowledge we were the first to mention the concept drift in software projects [Ekanayake et al., 2009]. However, we were failed to uncover any real activity that coincides with the periods where drift occurred. We can find many concept drift studies in other fields.

Vorburger and Bernstien [Vorburger and Bernstein, 2006] said that context changes can be treated like concept shifts, since the underlying data generator (the concept) changes while moving from one context situation to another. They presented an entropy based measure for data streams that is suitable to detect concept shifts in a reliable, noise-resistant, fast, and computationally efficient way. They assessed the entropy measure under different concept shift conditions and

illustrated the concept shift behavior of the stream entropy. They also presented a simple algorithm control approach to show how useful and reliable the information obtained by the entropy measure is compared to a ensemble learner as well as an experimentally inferred upper limit. Last but not least, they demonstrated the usefulness of the entropy based measure context switch indication in a real world application in the context-awareness/wearable computing domain.

Summarizing, we do not investigate the concept drift in software projects during this study. However, we believe that some observations of our experiments are due to the concept drift, but not yet proved. Further, the software systems behave similar to dynamical systems and the dynamical systems are sometimes subjected to concept drifts. Therefore, we propose that it is worth to investigate the concept drift of software projects in future researches.

2.8 Summery of Related Work

The goal of this thesis is to present temporal reasoning techniques to improve the reliability of prediction models in the Software Engineering domain and hence, this thesis is associated with the research field Data Mining and Software Engineering. The overview of the related work shows that a large number of prediction models have been developed for various activities in the Software Engineering. The ultimate goal of those models is to enhance the software quality. However, majority of those models are not being used for practical purposes since they are not reliable enough or not generalized properly. We started with reviewing the learning algorithms and their properties in Section 2.1. Further, we justified the AUC as the better performance metric of prediction models from evidence in the literature. We use similar techniques as [Zimmermann et al., 2007] to link Bugzilla and CVS repositories (see Section 2.2). Most of the features we used to train models are extracted from the literature. But, some features are defined by ourselves. We reviewed several prediction models and they used similar techniques to train and test their models. Further, they stick to the assumption that the underline data distribution is relatively stable over time. We found few publications about improving reliability of prediction models. They used a common approach, which is to combine/adapt a set of existing models. However, this is computationally expensive and not a trivial task. In our study we present temporal reasoning techniques, which is very simple to implement, for improving reliability of the models. The data quality is an important issue in the empirical software engineering researches. We took this matter seriously into consideration when preparing the software data for our experiments. We applied similar measurements as described in Section 2.6 to maintain the data quality. Finally, in Section 2.7 we discussed about Concept Drift since our experiment results shade a light about possible concept drifts in software projects. Unfortunately, in this study we are not able to find strong evidences to show the existence of concept drift in software projects.

Part II

Possible Threats for All Experiments

3

Threats to Validity

In this section we briefly discuss the most important threats to validity concerning the data gathering process, the data itself, and the applied methodologies. These threats are in general existing for all the experiments.

3.1 Determination of Authorship

We consider the committer to be the author of a change. This is possibly wrong when looking at projects that do not allow direct write-access to CVS. Apache's code base, *e.g.*, can only be changed via trusted proxy persons (*i.e.* committers). For our experiments, we did not consider source code information and therefore needed to rely on the information available from the versioning system (CVS). Even with the source code information available (*e.g.*, relying on the `@author` tag from Javadoc) we could not be sure that the listed person is also the author of the code change. Consider the following example: a developer makes a minor addition to the code in order to fix a defect and does not add himself to the list of authors in the source code because he thinks it is not worth mentioning. In such a situation the initial author of the file would be considered to also have made the bug-fix. Hence, our method is limited to determining the person who brought the code into the project's codebase – but that without doubt (be it as the actual author or not).

3.2 Creation-time vs. Commit-time

In this work we did only consider data that is made publicly available by the developer. Since we use a time-based partitioning of the datasets we make an implicit assumption that bugs opened at the moment when they are reported and are being fixed at the moment when a respective code change is committed. This may not always be correct because a code change may have been made long before committing (on the developers private workspace). Also, a bug might be in the code for months (or years) without being noticed. Given the available data we see no way to address this limitation. However, from a project management perspective it can be argued that defects and code changes only become relevant when they are reported. Only at reporting time they "materialize" as a task for the development team and cause further actions.

3.3 Bug-fixing or Enhancement? A Clear Case of Bias

It is hard to distinguish between bug fixing efforts and enhancements of the code (*e.g.*, the addition of new features or refactoring). Oftentimes, developers make a connection between a bug report and its related code changes by mentioning a bug ID in the commit message. However, this is a brittle connection without any mechanisms granting exclusivity of the submitted files to the mentioned bug report. A clear distinction between bug-fixing and code enhancement activities would require manually verified datasets (see also [Bachmann and Bernstein, 2009]). In addition to the brittle connection some information could be outright missing. For instance, a minor bug that is quickly fixed changes its state from open to fixed; without having a priority and/or severity assigned. Consequently, the data of this study clearly exhibits both commit feature bias as well as bug feature bias as introduced by [Bird et al., 2009]. In addition, [Ko and Chilana, 2010] revealed that most of power users reported non-issues that devolved into technical support, redundant reports with little new information and expert feature requests, and the reports that did lead to changes were reported by a comparably small group of experienced, frequent reporters. This implies that even the power users have no clear intention about the state of the reports.

Unfortunately, barring the availability of manually verified models, we see no practical way to address these biases. A main characteristic of the methods used in this work is the long-term evaluation of prediction models on software projects. To manually verify our datasets we would need to look into every bug report and every code change of a whole project and its history – an effort clearly beyond the scope of this study.

3.4 Missing fields

Some information is missing from versioning systems and bugtrackers. One reason could be a slack discipline of reporting (e.g. empty commit messages). It could also be that it is not necessary to fill those fields. For instance, a minor bug that is quickly fixed changes its state from open to fixed; without having a priority and / or severity assigned. Although, depending on the project, there are rules that demand complete reports, it is in many cases not feasible to maintain a complete bug database. We believe that such a phenomenon is of constant nature: either a project has missing fields or it hasn't. The reporting behavior only changes when new rules of reporting are applied and this does not happen often in a project. We assume that the amount and identity of missing fields are constant. Hence, if there are for instance many missing bug-report priorities, our learning algorithms will not pick those features for the prediction model since their predictive power is low.

Summarizing, the above mentioned threats are influencing for all of our experiments. Further, all of those issues associated with data gathering process. Though, we apply some approaches in the literature (*e.g.*, [Zimmermann et al., 2007]) to enhance the data quality the above mentioned drawbacks are still existing in the data. The only possibility to avoid all those issues is to use

manually verified dataset, which is practically very expensive process. However, we guaranty that those issues post minor threat for the experimental results.

Part III

Software Engineering Process Data

4

Explored Software Projects

We mainly use software engineering process data for all of our experiments conducted in this thesis. We explore four open source software (OSS) projects. They are large scale software projects with long development history.

(1) Eclipse[1]
(2) Netbeans[2]
(3) Mozilla[3]
(4) Open Office[4]

Many developers from all over the world are involving in the development process of these projects. Some of them are supporting to generate source codes for adding new features or fixing bugs and other are supporting to quality assessment or sending bug reports. Further, a large number of uses are using these software projects and they also helping developers by sending bug reports and requesting new features to enhance the projects. More importantly, these projects are widely used for empirical research in software engineering.
The source codes and the history information of the projects are freely accessible. These projects use Bugzilla or IssueZilla as Bugs Tracking Systems (BTS) and CVS or SVN as Version Control Systems.
Before discussing these projects in more details we provide a brief introduction about integrated development environment (IDE), VCS and BST, which are associated tools of the development process of those projects, in particular and for other software projects, in general.

4.1 Integrated Development Environment (IDE)

IDEs typically present a single program in which all development is done. This program typically provides many features for authoring, modifying, compiling, deploying and debugging software.

[1]http://www.eclipse.org/
[2]http://netbeans.org/
[3]http://www.mozilla.org/
[4]http://www.openoffice.org/

The aim is to abstract the configuration necessary to piece together command line utilities in a cohesive unit, which theoretically reduces the time to learn a language, and increases developer productivity. It is also thought that the tight integration of development tasks can further increase productivity. For example, code can be compiled while being written, providing instant feedback on syntax errors. While most modern IDEs are graphical, IDEs in use before the advent of windowing systems (such as Microsoft Windows or X11) were text-based, using function keys or hotkeys to perform various tasks (Turbo Pascal is a common example). However, an IDE is only for development purpose and it does not keep any information about development history.

4.2 Version Control Systems (VCS) and Bug Tracking Systems (BTS)

In the Software Engineering, version control is any practice that tracks and provides control over changes to source code. Software developers use VCS to maintain documentation and configuration files as well as source code. Specially, when more than one developer is engaging in developing the project and they are concurrently editing the project the VCS provides facilities to track all the changes done by the developers. Further, the VCS keeps information about the author-supplied commit message that usually states the reason for the change, author name, date, and lines of code changes in its log file. The Concurrent Version System (CVS) and Subversion (SVN) are widely used VCSs. Figures 4.1 and 4.2 show CVS and SVN change records (log files) respectively.

```
----------------------------
RCS file: /CVSroot/eclipse/org.eclipse.debug.core/core/org/eclipse/debug/core/commands/AbstractDebugCommand.java,v
Working file: org.eclipse.debug.core/core/org/eclipse/debug/core/commands/AbstractDebugCommand.java
Head 1.10
[.........]
Revision 1.10
Date 2005-05-18 14:32:15 +0100; author kian; state: EXP; lines: +6 -15
Bug 284363 -  Move DebugCommandAction to an API package
----------------------------
[.....]
----------------------------
Revision 1.1
Date 2002-01-10 12:30:35 +0100; author kim; state: EXP;
Bug 289263 - javadoc warning in N200909102000
============================================================================
```

Figure 4.1: CVS: log file

Bug tracking systems, on the other hand, record facts about known bugs. Facts may include the time a bug was reported, its severity, the erroneous program behavior, and details on how to reproduce the bug; as well as the identity of the person who reported it and any programmers who may be working on fixing it. Typical bug tracking systems support the concept of the life cycle (see Figure 4.3) for a bug which is tracked through status assigned to the bug. The BSTs

```
$ svn log
------------------------------------------------------------------------
r20 | harry | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line

Tweak.
------------------------------------------------------------------------
r17 | sally | 2003-01-16 23:21:19 -0600 (Thu, 16 Jan 2003) | 2 lines
```

Figure 4.2: SVN: log file

allow administrators to configure permissions based on status, move the bug to another status, or delete the bug. Some systems will e-mail interested parties, such as the submitter and assigned programmers, when new records are added or the status changes. Bugzilla and Issuezilla are popular BTSs.

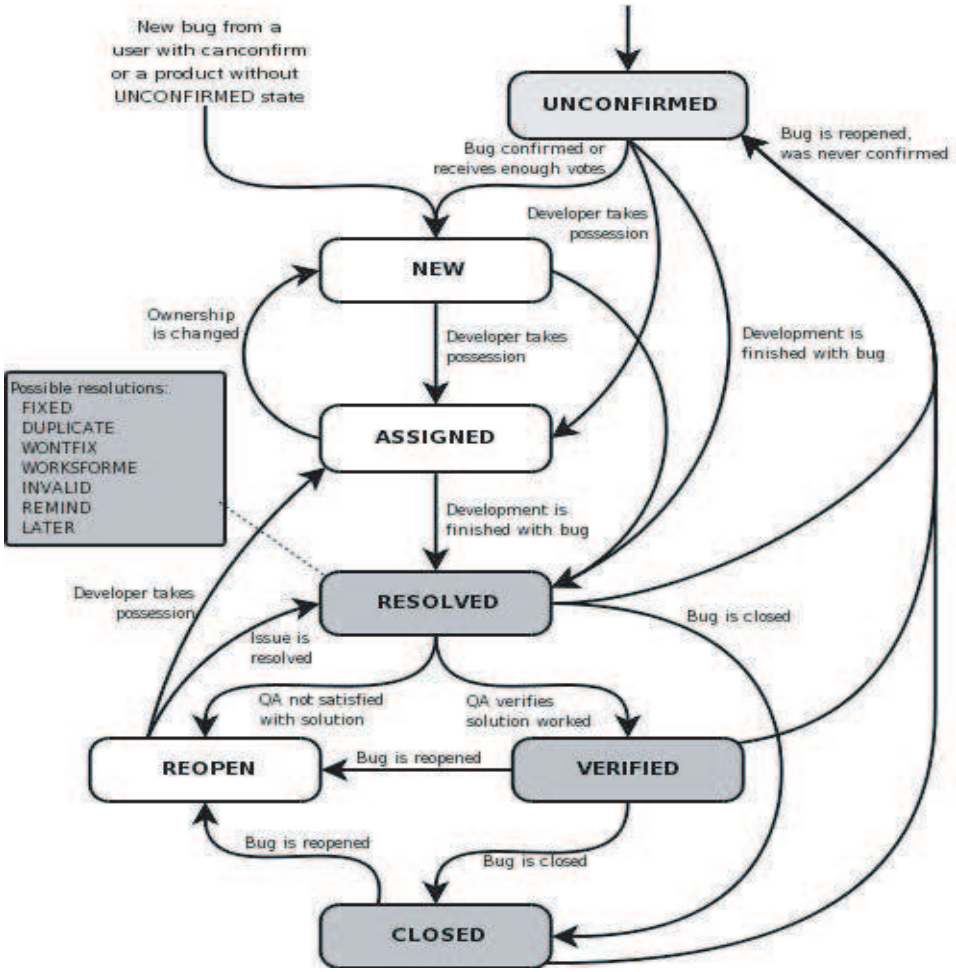


Figure 4.3: Bug life-cycle [The Bugzilla Guide - 2.18.6 Release]

4.3 Eclipse

Eclipse is an open source community, whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle. The Eclipse Foundation is a not-for-profit, member supported corporation that hosts the Eclipse projects and helps cultivate both an open source community and an ecosystem of complementary products and services.
The Eclipse Project was originally created by IBM in November 2001 and supported by a consortium of software vendors. The Eclipse Foundation was created in January 2004 as an independent not-for-profit corporation to act as the steward of the Eclipse community. The independent not-for-profit corporation was created to allow a vendor neutral and open, transparent community to be established around Eclipse. Today, the Eclipse community consists of individuals and organizations from a cross section of the software industry.
Eclipse integrated development environment (IDE) is very popular among software developers since it contains advance features and the extensible pug-in system, which are very helpful for them. It is primarily written in Java and can be used to develop Java applications, and with the help of other plug-ins, in other languages as well (such as C, C++, COBOL, Python, Perl, PHP *etc.*). Eclipse IDE mainly contains three sub-components; Platform, Plug-ins development environment (PDE) and Java development tools (JDT). These three sub-components contain several other components. In this thesis we use data from over 30 sub-components.
Further, Eclipse data is popular among the researches in the empirical software engineering domain. There are many publications based on Eclipse data in the literature [Čubranić and Murphy, 2003], [Zimmermann et al., 2007],[Bachmann et al., 2010],[Anvik, 2006], [Joshi et al., 2007], [Bettenburg et al., 2007], [Moser et al., 2008].

4.4 Netbeans

NetBeans started as Xelfi, which is a Java Integrated Development Environment (IDE), in Czech Republic in 1996. Xelfi was written in Java and was first released in 1997. Later Roman Stanek, an entrepreneur invested on Xelfi and produced the Netbeans commercially. Later, the Sun Microsystems bought Netbeans. Soon after the acquisition, Netbeans became open sourced. This was the Sun's first open source project. In June 2000, the *www.netbeans.org* web site was launched. Similar to Eclipse, Netbeans comprised with a large number of plug-ins that support for many applications such as Java, JavaScript, PHP, Python, Ruby, Groovy, C, C++, Scala and many more.

However, it is very seldom to use Netbeans's data for research projects and only very few publications available [Bachmann et al., 2010]. Nonetheless, we use Netbeans data since Eclipse and Netbeans have similar characteristics that facilitate us to compare experimental results.

4.5 Mozilla

The Mozilla project was created in 1998 with the release of the Netscape browser suite source code that was intended to harness the creative power of thousands of programmers on the Internet and fuel unprecedented levels of innovation in the browser market. Within the first year, new community members from around the world had already contributed new functionality, enhanced existing features and became engaged in the management and planning of the project itself. After several years of development, Mozilla 1.0, the first major version, was released in 2002. This version featured many improvements to the browser, email client and other applications included in the suite, but not many people were using it. In 2003, the Mozilla project created the Mozilla Foundation, an independent non-profit organization that continued the role of releasing software, such as Firefox – a web browser –, Thunderbird – an email client –, Bugzilla - a bug tracking tool –, Sunbird – a calender tool – and many more.
Mozilla project's data is often used for empirical software engineering research [Mockus, 2008], [Antoniol et al., 2008b], [Gyimothy et al., 2005], [Hooimeijer and Weimer, 2007], [Bettenburg et al., 2008]. The reason for including Mozilla for this investigation is to generalize the finding of the experiments on data other than IDEs.

4.6 Open Office

Oracle Open Office known before 2010 as StarOffice is Oracle's proprietary office suite software package. It was originally developed by StarDivision and acquired by Sun Microsystems in August 1999. The source code of the suite was released in July 2000 with the aim of reducing the dominant market share of Microsoft Office by providing a free and open alternative; later versions of StarOffice are based upon OpenOffice.org with additional proprietary components. The OpenOffice.org project is primarily sponsored by Oracle Corporation (having acquired Sun Microsystems). Other major corporate contributors include Novell[5], Red Hat[6], IBM[7], Google[8]. In this thesis we use data only from SW component, which is the writer application of Open Office suite.
There are many research publications [Bachmann et al., 2010], [Canfora and Cerulo, 2005], [Koru and Tian, 2005], [Bakota et al., 2006] based on Open Office project's data.

[5]http://www.novell.com/
[6]http://www.redhat.com/
[7]http://www.ibm.com/
[8]http://www.google.com

4.7 Pre-Existing Datasets

We created our own datasets from above four projects. We used approaches explained in the literature to develop these datasets. However, there are pre-existing Eclipse datasets[9], which is developed by [Zimmermann et al., 2007]. Later, this dataset was further enhanced by Nagappan *et al.* [Nagappan et al., 2010] and placed in the same URL. These datasets have been used in many other people [Moser et al., 2008], [Čubranić et al., 2005] for researches. However, for our experiments we need temporal based data and hence, we are not able to use these datasets.

4.8 Conclusion

In this section we first, discussed about the software engineering tools; IDEs, VCSs and BTSs. Next, we discussed about the explored projects and their histories. We provided the reasons for using these projects' data for our investigations. Finally, we presented the existing datasets in the literature for empirical researches. In the next chapter we provide detail description of each of the features extracted from the above software projects.

[9] http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/

5

Feature Description

In this section we describe the features that we use for defect and cost prediction models. As we mentioned in the subsection 2.2.4, we first search for features that frequently used in the past researches. Furthermore, we generate new features using our background knowledge. Here, we provide the description of the base set of features. Depending on experiments, we construct additional features by combining these features. We will provide additional context on such features whenever it necessary.

5.1 Features: Defect Prediction

`revision`:
We consider a revision as a change made to a file for some reason. The feature `revision` represents the number of changes made to a file during training periods. Both [Graves et al., 2000] and [Khoshgoftaar et al., 1996] found that past changes are good defect indicators.

`activityRate`:
This feature measures how often a file has been revised during the training periods and is computed by dividing the number of revisions during the training period by the length of the training period (in months). [Hassan and Holt, 2005] concluded that a high frequency of changes in a file is a good defect predictor.

`lineAdded`, `lineDeleted` and `totalLineOperations`:
Several studies showed that past changes are good defect predictors ([Graves et al., 2000],[Khoshgoftaar et al., 1996]). Therefore, we further quantify the amount of change done by authors using the features `lineAdded` and `lineDeleted` that describe the number of lines of code added and deleted during training periods. Further, we introduce the total amount of work done for a revision by adding those two features resulting the feature `totalLineOperations`.

`grownPerMonth`:
This feature provides information about the growth rate of a project or file in the training periods.

We compute the amount of grown using the total number of line added and deleted during that time period. Usually, we subtract the total number of line deleted from the total number of line added and then average this value by dividing this number by the length of the training period (in months). Therefore, this number can be ether positive (representing growth) or negative (representing shrinkage). We introduced this feature to address issues that may arise due to too fast change.

`lineOperationRRevision`:
This feature captures the average size of a revision in terms of number of lines of code added and deleted. We simply add the total numbers of lines of code added and deleted during training periods and divide that amount by the number of revisions during that period.

`chanceRevision` and `chanceBug`:
These two features provide the probability of having a revision or a bug in a file in the future. These features mimic the award winning `BugCache` approach ([Kim et al., 2007]), which proposes that more recently fixed files are more vulnerable for bugs. We model this probability using the formula $1/2^i$, where i represents how far back (in months) the latest revision or bug occurred from the prediction time period. If the latest revision or bug occurrence is far from the prediction time period, then i is large and the overall probability of having a bug (or revision) in the near future is low.

`blockerFixes, criticalFixes, majorFixes, minorFixes, normalFixes` and `trivialFixes`:
These six features report the number of different types of bugs fixed during training periods. The bugs are categorized according to their severity such as blocker, critical, major, minor, normal and trivial. We can find the severity information of fixed bugs from bugzilla database. If a revision has a referenced or linked entry in the bugzilla database and the severity of that entry is marked as one of the above categories then we consider that the revision is for a bug fixing activity. Further, the bug-fixing revision date falls into the training periods then we count as one bug has been fixed in the assigned category. Our intention of introducing these features is to uncover any correlation between the severity and defects.

`enhancementFixes`:
This feature counts the number of revisions made for enhancements requested during the training period of the models. In the bug categorization process, authors find that some requests are not for fixing bugs, but for enhancements. Hence, we introduce the feature `enhancementFixes` that counts such fixed enhancements.

`blockerReported`, `criticalReported`, `majorReported`, `minorReported`, `normalReported` and `trivialReported`:
These six features provide information about the number of reported bugs in terms of severity. We introduce these features as not all reported bugs during a training period may be fixed within that period. Note that we consider the opening date of a bug and the reported date are same. If an opening date falls into the training period then we count as one bug has been reported in the assigned category.

`enhancementReported`:
This feature counts number of enhancements reported during training periods. The reported is determined as above.

`p1-fixes`, `p2-fixes`, `p3-fixes`, `p4-fixes` and `p5-fixes`:
Each Bug report is further categorized based on its priority such that the highest and the lowest priority bugs are categorized as P1 and P5 respectively . The other priorities are fallen in between P1 and P5. Theses five features describe the number of priority wise bugs fixed during training periods. Bug fixing dates are determined as in the above cases. If a bug-fixing date falls into the training periods then we count as one bug has been fixed in the assigned category.

`p1-reported`, `p2-reported`, `p3-reported`, `p4-reported`, and `p5-reported`:
These five feature provide information about the number of bugs reported with corresponding priority during training periods. The reported dates are determined as in the above.

`lineAddedI`, `lineDeletedI` and `totalLineOperationsI`:
Theses three features provide information about lines of code added, deleted, and total lines of code operated (or changed) to fix bugs during training periods. If a revision has a referenced entry or link in the bugzilla database and the corresponding bug report is not marked as an enhancement but has a severity levels then we consider that revision to be a bug fixing activity. Furthermore, the information in the CVS log allows us to extract how many lines of code where added and deleted for that revision supplying the basis for `lineAddedI` and `lineDeletedI`. Adding these two features results in `totalLineOpertaionsI`. These three variables capture how much work (in terms of number of lines of code) is accomplished by the authors to fix bugs.

`lineOperationIRBugFixes`:
This feature measures the average number of lines of code changed to fix bugs during the training periods. Thus, this features captures the size of the bugs fixed and provides any correlation between the average size of fixed bugs and the defects. This feature can be derived by dividing the total number of lines changed to fix bugs by the total number of bugs fixed.

`lineOperationIRTotalLines`:
This feature describes the work effort by the authors to fix bugs relative to the other work during the training periods. We already computed the total number of lines changed (or operated) to fix bugs and other activities such as enhancements. Hence, we can derive this feature by dividing the total number of lines to fix bugs by the total number of lines changed for any other activity.

`lifeTimeBlocker`, `lifeTimeCritical`, `lifeTimeMajor`, `lifeTimeMinor`, `lifeTimeNormal` and `lifeTimeTrivial`:
These six feature describe about the lifetime of different types of bugs fixed during training periods. Both Bugzilla and CVS databases provide the information about opening and closing dates of the bugs. Further, Bugzilla provides the severity level of a bugs. Consequently, we can compute the lifetimes of any type of bug by taking the difference between the closing and the opening dates. Note that even when the opening dates lie outside the considered training periods we use them to compute the bug lifetimes.

`hasBug`:
This is the target variable for J48 model. This variable describes whether any kind of bug (blocking, critical, major, minor, normal, or trivial) has been reported or not in target periods.

`hasNumberOfBug`:
This is the target variable for regression models (Both linear and non-linear). This describes how many bugs (blocking, critical, major, minor, normal, or trivial) has been reported during the target periods.

The above software attributes represent almost all the changes done for a file in the history.

5.2 Features: Cost Estimation

In addition to the software data we use cost estimation data to train models for predicting expected cost for a given project. However, our intention of this experiment is to uncover the influence of the time factor on cost estimation prediction. We use Repository Data Release-9 of the International Software Benchmarking Standards Group (ISBSG). The attributes of this dataset are defined by ISBSG.
Feature selection is done using the same methods described in the above section. Following is the description of each attribute that we use for cost prediction.

Data quality rating attributes
`DataQualityRating`: This field contains an ISBSG rating code of A, B, C or D applied to the project data by the ISBSG quality reviewers to denote the following:

A= The data submitted was assessed as being sound with nothing being identified that might affect its integrity.
B= The submission appears fundamentally sound but there are some factors which could affect the integrity of the submitted data.
C= Due to significant data not being provided, it was not possible to assess the integrity of the submitted data.
D= Due to one factor or a combination of factors, little credibility should be given to the submitted data.

Sizing attributes
`CountApproach`: A description of the technique used to size the project. For most projects in the ISBSG repository this is the Functional Size Measurement Method (FSM Method) used to measure the functional size (e.g. IFPUG, MARK II, NESMA, COSMIC-FFP etc.). For projects using Other Size Measures (e.g. LOC etc.) the size data is in the section Size Other than FSM. Helps you to compare apples with apples.

`AdjustedFunctionPoint`: For IFPUG, NESMA and MARK II counts this is the adjusted size (the functional size is adjusted by a Value Adjustment Factor). The resultant adjusted size is reported in adjusted function points (AFP). Where the Adjusted Size has not been supplied by the project then the Functional Size is used in the calculations that use AFPs.

Grouping attributes
`DevelopmentType` : This field describes whether the development was a new development, enhancement or re-development.

`BusinessAreaType`: This identifies the type of business area being addressed by the project where this is different to the organization type. (*e.g.,* Insurance, Finance, Manufacturing, Banking, Accounting, Other, ITT, Logistics, Health, SalesMarketing, Government).

`PackageCustomization`: This indicates whether the project was a package customization. Possible values for the attribute **PackageCustomization** are yes, no or don't know.

Project attributes
`DevelopmentPlatform`: Defines the primary development platform, (as determined by the operating system used). Each project is classified as: PC, Mid Range, Main Frame or Multi platform.

`LanguageType`: Defines the language type used for the project: e.g. 3GL, 4GL, Application Generator etc.

`CASEToolUsed`: Whether the project used any CASE tool. The full repository holds a breakdown

of CASE usage for those projects that reported using a CASE tool:
(1) Upper CASE tool
(2) Lower CASE tool with code generator
(3) Lower CASE tool without code generator
(4) Integrated CASE tool

`MainOperatingSystem`: This is the primary technology operating system used to build or enhance the software (i.e. that used for most of the build effort).

`DevelopmentProcessModel`:This defines the development process model used by the development team to build the software. (*e.g.,* waterfall, iterative).

`HowMethodologyAcquired`: This describes whether the development methodology was purchased or developed in-house, or a combination of these.

Product attributes
`BusinessUnits`: This represents the number of business units (or project business stakeholders) serviced by the software application.

`Locations`: This represents the number of physical locations being serviced/supported by the installed software application.

`ConcurrentUsers`: This represents the number of users using the system concurrently.

`IntendedMarket`: This field describes the relationship between the project's customer, end users and development team.

Effort attribute
`AverageTeamSize`: This field describes the average number of people that worked on the project. This is calculated from the team sizes per phase.

Target variable
`SummaryWorkEffort`: This is the target attribute and it provides the total effort in hours recorded against projects.

We use above attributes (both in defect and cost prediction domains) to train prediction models for achieving our main goal of this thesis.
In the next section we provide the composition of the defect datasets for training models.

5.3 Composition of Defect Datasets

A defect dataset contains two parts, labeling and feature computation. The length of the labellings period is usually one month and in this period, we record the number of bugs reported – target variable – for each observed file. The length of the feature computation period – training period – can be extended from one month to the maximum length of the observed period and further, this period starts one month before the labeling period and expands into past. In this period, we compute the above mentioned features (depending on the experiment) of each file, which we recorded the number of bugs reported during the labeling period. Following is the mathematical notion of a dataset for most of the experiments. But, some experiments use slightly different format of the datasets and we will define when they are differ from the following notion.

Assume that the observed period is d months. $Y_T = \{y_{T,1}, y_{T,2}, ..., y_{T,j}, ..., y_{T,s}\}$ is a vector of dimension s (s is the number of observed files) and $y_{T,j}$ is the number of bugs reported for file j at time T, where $1 < T \leq d$. if $X_t = \{f_{t,1}, f_{t,2}, , f_{t,i}, ..., f_{t,n}\}$ is a feature vector of dimension n, where $n \in N$ and $f_{t,i}$ is a file feature i computed from the history information at time t, where $1 < t \leq d-1$, $t < T$ and $s >>> n$, then constructed dataset is given by $\bigcup_{t=x}^{T-1} X_t, Y_T$, where x is beginning of the training period. By changing the x and T variables we can generate different datasets.

Summarizing, in this chapter we provided a detail description of features that we use for the experiments. Furthermore, we described the composition of the datasets that are used for training defect prediction models.

Part IV

Time-Based Information Influences Defect Prediction

6

Impact of time on prediction quality

Simplest hypothesis is the best
14th century English philosopher
William of Ockham

The core contribution of this thesis is to bring the temporal reasoning in predicting defects of software systems. To that end, we provide novel techniques based on temporal reasoning for improving the defect prediction quality that ultimately contributes to software quality. These techniques should complement already existing quality prediction models. Part I of this thesis has already motivated the need for these novel techniques to improve the reliability of prediction models by presenting their common drawbacks. We have presented the most important related work, and have reviewed that there is a relationship between real-time and evolving rules of systems. Therefore, understanding and modeling this relationship is the key to describe the behavioral factors of the systems in future. In this chapter, we design experiments to address the 1st subproblem mentioned in the Section 1.2 – finding the impact of real-time based information on prediction quality.

6.1 Preliminaries

In the introduction section, we mentioned that the most prediction models have not properly taken the time factor into consideration. The issue is:
"`Is there any significant effect on prediction quality regardless of the time factor?`".
In this chapter, we design an experiment to address the above issue. Further, if the time factor has such significant influence on the prediction quality then this can be considered as the gate way for the rest of the thesis. Therefore, we first define the two data sampling techniques: temporal and random sampling.
`Temporal Sampling Technique:`
In this sampling technique, datasets are always collected for training models during specific periods – training periods – prior to prediction periods – target periods. This is the typical data sampling technique and it reflects that history or information prior to the prediction periods is

better predictors.

`Random Sampling Technique:`

In contrast to the above sampling method, in this sampling method the dataset is formed for training the model using the data collected randomly before and after the prediction period. The rational behind this sampling technique is that we do not take the time factor into consideration during data collection.

To our knowledge there is no formal investigation about the impact of the above sampling techniques on prediction quality. Therefore, we conduct a comparison study between these two sampling techniques on prediction quality. Further, this study is carried out using the data collected from two important areas; defect prediction and cost estimation in the software engineering domain. Typically, an experiment is designed to test hypotheses. Therefore, in this experiment, we test the following hypotheses.

H 1.1: *In the defect prediction domain, models trained on data collected from the temporal sampling technique are better than models trained on data collected from the random sampling technique.*

H 1.2: *In the cost estimation domain, models trained on data collected from the temporal sampling technique are better than the models trained on data collected from the random sampling technique.*

Next, we explain the composition of datasets that we use for the experiment, prediction model and its performance measures.

6.1.1 Data Description: Eclipse, Netbeans and Cost Estimation

We use the Eclipse and Netbeans projects data for this experiment and data collection and feature generation are as mentioned in the Section 2. Further, we have already explained the reason for selecting these two projects in the Section 2. The observed period for data collection is from March 01, 2004 to March,01 2008 and the development history of four years is sufficient for this kind of analysis to ensure the gathering of multiple development cycles. The Netbeans repository has been shifted from Concurrent Versions System (CVS)[1] to Subversion (SVN) [2]on March 2008. The SVN does not provide information about the lines of code operations of files in each revision. However, our models use this information for predicting defects. Hence, we collect Netbeans data until March 2008. We use the same observed period for Eclipse as it is fair and easy to compare the results. As we mentioned in the Section 2 we use only unique file names and source code file type *.java in both projects and not marked as "Dead" in the observed period.

The composition of datasets is as mentioned in the Section 5. However, we provide a formal definition of the dataset because the composition of this dataset is slightly differ from the composition of the the dataset mentioned in the Section 5.

[1]http://www.nongnu.org/cvs/

[2]http://subversion.apache.org/news.htmlnews-20100217

Assume that the observed period is divided into $2, 3, ..., d$ *months, where* d *is the maximum length of the observed period in months. In each partition,* $Y_T = \{y_{T,1}, y_{T,2}, ..., y_{T,j}, ..., y_{T,s}\}$ *is a vector of dimension* s *(* s *is the number of observed files) and* $y_{T,j}$ *is the number of bugs reported for file* j *at time* T*, where* $1 < T \leq d$ *and* T *is one-month long labeling period . if* $X_t = \{f_{t,1}, f_{t,2}, , f_{t,i}, ..., f_{t,n}\}$ *is a feature vector of dimension* n *and* $f_{t,i}$ *is a file feature* i *computed from* t*-months long training period, where* $n \in N$ *and* $1 \geq t \leq d-1$*,* $t < T$ *and* $s >>> n$*, then constructed dataset is given by* $\bigcup_{t=x}^{T-1} X_t, Y_T$*, where* x *is the beginning of the training period*.

We use features listed in Table 6.1 to train models. The detailed description of each feature can be found in the Section 5. However, for the readers convenience we provide a brief description of each feature of defect dataset in the following paragraph.

6.1.2 The Data: Features

The features listed in the Table 6.1 reflect information about or changes made to a file in the past. All the features, with the exception of the target variable `hasBug`, are computed during the training period of a model. The target variable is computed in the labeling period. The training period always follows the labeling period with no overlap, as in, any realistic situation the target variable `hasBug` is the only known ex-post (*i.e.* in the future) whilst the other features are available ex-ante (*i.e.* at the time where a prediction is made).

Target feature: The target feature describes the number of bugs that have been reported for each file during the labeling period. The bug reported date is determined as described in the Section 2.

Other features Most of the names of the features listed in Table 6.1 are self-explanatory. Nevertheless, a complete description can be found in the Section 5. However, we briefly discuss about some of these features as follows:

The `activityRate` represents the number of activities (revisions) took place per month. To determine the rate we count the number of revisions during the training period and then divide it by the length of the training period (leading to an averaging of the value). The length of the training period is given in months.

The features `lineAdded` and `lineDeleted` are the total number of lines of code added and deleted in all revisions during the training period. The sum of the above two features (`lineAdded` and `lineDeleted`) is `totalLineOperations`.

`grownPerMonth` describes the evolution of the overall project (in terms of lines of code) in the training period. Specifically, we compute the difference between the number of lines added and deleted (*i.e.* `lineAdded` and `lineDeleted`). This number can be positive (growth) and negative (shrinkage). We then average this value by dividing it by the length of the training period.

The feature `lineOperationRRevision` describes the average number of lines added and deleted per revision during the training period.

Name	Description
Features only from Versioning System (per file)	
activityRate	Number of revisions per month
lineAdded	# of lines added
lineDeleted	# of lines deleted
lineOperationRRevision	Number of line added and deleted per revision
revision	Number of revisions
totalLineOperations	Total # of lines added and deleted
Features only from Bugtracker	
blockerFixes	# of blocker type bugs fixed
blockerReported	# of blocker type bugs reported
criticalFixes	# of critical type bugs fixed
criticalReported	# of critical type bugs reported
enhancementFixes	# of enhancement requests fixed
enhancementReported	# of enhancement requests reported
majorFixes	# of major type bugs fixed
majorReported	# of major type bugs reported
minorFixes	# of minor type bugs fixed
minorReported	# of minor type bugs reported
normalFixes	# of normal type bugs fixed
normalReported	# of normal type bugs reported
trivialFixes	# of trivial type bugs fixed
trivialReported	# of trivial type bugs reported
p1-fixes	# of priority 1 bugs fixed
p1-reported	# of priority 1 bugs reported
p2-fixes	# of priority 2 bugs fixed
p2-reported	# of priority 2 bugs reported
p3-fixes	# of priority 3 bugs fixed
p3-reported	# of priority 3 bugs reported
p4-fixes	# of priority 4 bugs fixed
p4-reported	# of priority 4 bugs reported
p5-fixes	# of priority 5 bugs fixed
p5-reported	# of priority 5 bugs reported
Features from both CVS and Bugtracker	
lineAddedI	# of lines added to fix bugs
lineDeletedI	# of lines deleted to fix bugs
lineOperationIRbugFixes	Average number of lines operated to fix a bug
lineOperationIRTotalLines	# of lines operated to fix bugs relative to total line operated
lifeTimeBlocker	Average lifetime (avg. lt.) of blocker type bugs
lifeTimeCritical	avg. lt. of critical type bugs
lifeTimeMajor	avg. lt. of major type bugs
lifeTimeMinor	avg. lt. of minor type bugs
lifeTimeNormal	avg. lt. of normal type bugs
lifeTimeTrivial	avg. lt. of trivial type bugs
totalLineOperationsI	Total # of lines touched to fix bugs
grownPerMonth	Project grown per month (can be negative)
Target feature	
NumberOfBug (Target)	Indicates the number of bugs in labeling period

Table 6.1: Extracted variables (features) from software data

The features from `blockerFixes` to `p5-reported` provide information about the different types of bugs reported and fixed for files during the observed training period. If an opening date of a bug reported for a file falls into the training period, then a bug is considered as being reported during the training period. Analogously, we count the number of fixed bugs.

`lineAddedI`,`lineDeletedI` and `totalLineOperationsI` provide the number of lines operated to fix bugs and `lineOperationIRbugFixes` provides the average number of lines operated per bug.

`LineOperIRTolLines` counts how many lines were added and deleted to fix bugs in relation to the total number of lines added/deleted. This indicates what fraction of changes is focused on fixing bugs in relation to other activities (such as adding new features).

Finally, the remaining features represent the average lifetimes of bugs. Note that if a bug fixed is revised then the revision date is considered as the closing date of that bug. The corresponding entry for the bug fixing revision in the bug database provides the opening date of the bug and, hence, we can compute the lifetime of a bug. When the opening date lies outside the training period we still use it to compute the lifetime of the bug.

In addition to the defect prediction data we use the ISBSG Repository Data Release-9 for cost prediction. The repository contains information about 2842 projects. This information contains the starting year, properties of each project and effort that provides the total effort spent in hours of each project. Table 6.2 lists all the features and a description of each feature can be found in the Section 5. The observed time period is from 1989 to 2004. Unlike the defect prediction datasets, the cost prediction dataset does not contain two parts. All the features listed in the Table 6.2 are computed in the same time frame. But, we define effort spent for each project as the target or the dependent variable and the rest as the predictors or the independent variables. Further, the length of the training period is varied from 1 year to the maximum length of the observed period.

6.1.3 Choice of Algorithm and Performance Measurements

In this experiment, we use simple linear regression as the prediction algorithm. This algorithm is implemented in WEKA data mining framework [Witten and Frank, 2005].

There is a formal reason for selecting simple linear regression model. Following the Ockhams razor argument – simplest is usually the best one. Our goal is to predict the expected number of bugs for a file and the summery work effort for a given project using the above datasets. The prediction quality is measured using the Pearson correlation between the true and predicted values, mean absolute error (MAE) and root mean square error (RMSE). The reason behind using Pearson correlation is the fact that predicted values and the actual values show approximately normal distribution (using normal probability plot).

Name	Description
Rating Attribute	
DataQualityRating	This feature contains an ISBSG rating code
Sizing Attribute	
CountApproach	A description of the technique used to size the project
AdjustedFunctionPoint	For IFPUG, NESMA and MARK II counts this is the adjusted size
Grouping Attributes	
DevelopmentType	This field describes whether the development was a new development, enhancement or re-development
BusinessAreaType	This identifies the type of business area being addressed by the project
PackageCustomisation	This indicates whether the project was a package customization
Project Attributes	
DevelopmentPlatform	This defines the primary development platform
MainLanguageType	This defines the language type used for the project
CaseToolUsed	This defines whether the project used any CASE tool
MainOperatingSystem	This defines which operating system is used to develop the project
DevelopmentProcessModel	This defines which methodology is employed by the developers
HowMethodologyAcquired	This describes whether the development methodology was purchased or developed in-house, or a combination of these
Product Attributes	
BusinessUnits	Number of business units serviced by the software application
Locations	Number of physical locations being serviced/supported by the installed software application
ConcurrentUsers	Number of users using the system concurrently
IntendedMarket	This field describes the relationship between the project's customer, end users and development team
Effort Attributes	
AverageTeamSize	The average number of people that worked on the project
Target Variable	
SummaryWorkEffort	This provides the total effort in hours recorded against projects

Table 6.2: Extracted variables (features) from cost estimation data

6.2 Method Implementation

In the Section 6.1 we briefly discussed the temporal and random sampling techniques. However, in this section, we provide exemplary implementation of those two methods. We use the WEKA APIs inside a Java Development Environment to implement the experiment. In the next paragraph, we discuss the implementation of the temporal sampling experiment in details.

In this experiment we always train prediction models from the data collected prior to the prediction period – target period. The datasets construction are as mentioned in the Section 6.1.1. Initially, we use defect data to conduct the experiment.

First, we train a model using the data collected from two months (the minimum length of the training period is two months) before the target period. The initial target period is the third month from the beginning of the observed period. Next, we move the target period by one month in order to expand the training period and repeat this process until we reach the maximum observed period. For example if the initial target period is May-2004 then the initial training period is March-April 2004 (March is the feature computation period and April is the labeling period). We then move the target period one month forward and that brings the target period into June-2004 and the training period into March-May 2004. In each model, we measure its prediction quality using correlation coefficient between the true and predicted values, MAE and RMSE.

Next, we conduct the same experiment using the cost prediction data. The experimental procedure is same as above. However, in this experiment, we start predicting the expected cost of projects implemented in the year 1990 (target period) using the model trained from the data collected in 1989 (training period). And then move the target period by 1 year in order to expand the training period. This procedure is repeated until we reach the maximum observed period. Similar to the above experiment the models' prediction quality is measured using correlation coefficient, MAE and RMSE.

Our next step is to conduct the random sampling experiment using defect and cost estimation data. First, we use the defect data to conduct this experiment. We divide the observed period into many time windows. The Initial length of a time window is two months (the minimum training period is two months). Among these time windows, we consider a time window as a target period and collect data randomly from every other time windows (except the target period) in order to train a prediction model. The number of data points collected is similar to the initial sample size of the temporal sampling technique. The model's prediction quality is measured using correlation coefficient, MAE and RMSE. This procedure is repeated considering each time window as a target period. In the initial run we trained 46 prediction models. We measure the prediction quality of each of these models using correlation coefficient, MAE and RMSE values and then compute the mean and median of the correlation, MAE and RMSE values to estimate the average prediction quality of all those 46 models. Now we can compare the prediction quality in terms of

correlation coefficient, MAE and RMSE between the temporal and random sampling techniques (the length of the training period is two months). The above procedure is repeated by dividing the observed period into time windows, which are 3,4,5,....,n-1 months, where n is the maximum length of the observed period.

Next, we conduct the same experiment using the cost estimation data. However, the composition of the cost estimation data is differ from the defect prediction data. Therefore, in this experiment we allocate 01 year time period for a window. As in the above experiment we select data randomly from every time window except the target year to train prediction models.

We compute the prediction quality of both random and temporal sampling techniques for every length of the training periods (2 to 45 months for the defect data and 1 to 15 years for the cost estimation data). Our next step is to compare the prediction quality between these two sampling techniques. Please note that the comparison is taken place between the models trained on similar training lengths.

6.3 Results and Discussion

In this section we discuss the comparison results of the defect prediction and the cost estimation domains. We first present the result of the defect prediction using Eclipse and Netbeans data.

Temporal Vs. Random Sampling in Defect Prediction: Eclipse Data

The comparison is taken place between the random and temporal sampling techniques. For example: the correlation of temporal sampling is compared with the mean and median correlation of random sampling. In order to compare two datasets, selecting the right test is very impotent. Some of the tests are parametric while others are non-parametric. Hence, we first check the distribution (normal) of each variable (Correlation, MEA and RMSE) using *One-Sample Kolmogorov-Smirnov* (K-S test) test. Table 6.3 lists the $P - value$ at $\alpha = 0.05$ level (at 95% of confidence interval).

	Temporal	Random	
		Mean	Median
Correlation	0.205	0.943	0.25
MAE	0.389	0.083	0.029
RMSE	0.625	0.001	0.003

Table 6.3: Eclipse: One-Sample Kolmogorov-Smirnov (K-S test) test for normality

According to the Table 6.3, the median of MAE and, the mean and median of RMSE in random sampling technique are not from a normal distribution. Therefore, we cannot use any parametric test such as paired t-test for comparing these datasets. We hence, use non-parametric test *Wilcoxon*

Signed Ranks for comparing these datasets.

Tables 6.4 and 6.5 show the prediction quality comparison between the random sampling and temporal sampling techniques using paied t-test and Wilcoxon Signed Ranks test respectively. In these tables, "Pair" column indicates the variable pair that we are going to compare, "Mean" column indicates the mean of each variable and "Sig." column indicates the p-values at $\alpha = 0.05$ level (at 95% of confidence interval).

Pair	Mean	Sig.
Correlation-temporal Correlation-random (mean of correlation values)	0.21 0.19	0.037
Correlation-temporal Correlation-random (median of correlation values)	0.21 0.18	0.006
MAE-temporal MAE-random (mean of MAE values)	0.75 0.78	0.003

Table 6.4: Eclipse: Comparison results using paired t-test

Pair	Mean	Sig.
MAE-temporal MAE-random (median of MAE values)	0.75 0.77	0.036
RMSE-temporal RMSE-random (mean of RMSE values)	0.92 0.91	0.35
RMSE-temporal RMSE-random (median of RMSE values)	0.92 0.90	0.21

Table 6.5: Eclipse: Comparison results using Wilcoxon Signed Ranks

Temporal Vs. Random Sampling in Defect Prediction: Netbeans Data

Before applying any comparison test we conduct the test for normal distribution – K-S test– as in the above case. Table 6.6 lists the test results (significant values at $\alpha = 0.05$ level (at 95% of confidence interval)).

	Temporal	Random	
		Mean	Median
Correlation	0.978	0.003	0.00
MAE	0.747	0.24	0.031
RMSE	0.169	0.000	0.000

Table 6.6: Netbeans: One-Sample Kolmogorov-Smirnov test for normality

According to the K-S test only MAE of temporal sampling technique and the mean of MAE of random sampling technique (these two variables are following normal distribution) can be compared using a parametric test and the other datasets have to be compared using a non-parametric

test. To that end we apply Wilcoxon Signed Rank test for comparing datasets, which do not follow normal distribution and paired t-test for the other datasets. Tables 6.7 and 6.8 display the comparison results.

Pair	Mean	Sig.
Correlation-temporal Correlation-random (mean of correlation values)	0.25 0.22	0.008
Correlation-temporal Correlation-random (median of correlation)	0.25 0.22	0.006
MAE-temporal MAE-random (median of MAE values)	0.42 0.44	0.007
RMSE-temporal RMSE-random (mean of RMSE values)	0.57 0.55	0.1
RMSE-temporal RMSE-random (median of RMSE values)	0.57 0.54	0.035

Table 6.7: Netbeans: Comparison results using Wilcoxon Signed Ranks test

Pair	Mean	Sig.
MAE-temporal MAE-random (mean of MAE values)	0.42 0.44	0.004

Table 6.8: Netbeans: Comparison results using paired t-test test

Temporal Vs. Random sampling using cost estimation data

In this section, similar to the above section we compare the prediction results between the random and temporal sampling techniques. However, for this experiment we used cost estimation data. Like in the above section we first check the normality of each variable (correlation, MAE and RMSE) before applying any non-parametric test. Table 6.9 lists the normality result (using One-Sample Kolmogorov-Smirnov). In this table, "Temporal" and "Random" columns show the $P - values$ of the test in the two sampling techniques.

	Temporal	Random
Correlation	0.683	0.947
MAE	0.453	0.680
RMSE	0.239	0.393

Table 6.9: Cost estimation data: One-Sample Kolmogorov-Smirnov test for normality

According to the Table (6.9) all the variables are from a normal distribution. Therefore, we can apply independent sample t-test for the comparison study. Table 6.10 describes the compar-

ison result of the independent sampling t-test. In this table, "Sampling method" represents the sampling technique (temporal or random), "N" represents the sample size and "Sig." column represents the $P - value$ of the test. "Mean" and "Std. Deviation" represent the distribution of the variables.

	Category	N	Mean	Std. Deviation	$Sig.$
Correlation	Temporal	16	0.52	0.19	0.924
	Random	16	0.51	0.14	
MAE	Temporal	16	5758.7	3370.5	0.866
	Random	16	5934.50	2789.01	
RMSE	Temporal	16	12371.0	13272.09	0.562
	Random	16	10248.02	5773.31	

Table 6.10: Cost estimation data: Comparison result using independent sample t-test

The comparison results in the Tables 6.4, 6.5, 6.7 and 6.8 show that the correlation value – between the actual and observed – of the temporal sampling technique is significantly higher than the mean and median correlation of random sampling technique in predicting defects. Further, the MAE of temporal sampling technique is significantly smaller than the mean and median of MAE of random sampling technique, which implies the models trained on temporal sampling data make less error than models trained on random sampling data in predicting defects. However, the RMSE of temporal and random sampling have no significant difference. One reason for such behavior could be outliers that contribute heavily for RMSE in temporal sampling method. So, the above observations support the fact that in general, the temporal sampling obtains better prediction quality than random sampling in defect prediction domain. The Eclipse project has been developed over the last 10 year and the project has evolved chronologically. The same files or subsystems have been revised several times. The nature of one revision of a file or a subsystem influences its next revisions and also, there is always a time gap between two revisions. Therefore, for predicting defects, the information prior to the prediction time is superior to the information selected randomly without considering the time.

The comparison results of cost estimation indicate (see Table 6.10) that there is no significant difference between the random and temporal sampling methods. Contrast to the defect prediction data, the cost estimation data contains only the final cost of each project and there is no information about the cost of each revision or stage through out the development life cycle of a project. Hence, the relationship between the cost and time cannot be captured from this data and therefore, the models' prediction quality between the temporal and the random sampling techniques does not differ significantly.

6.4 Concluding Discussion

In this chapter we present an approach to investigate the influence of temporal and the random data on both defect and cost estimation prediction quality. We observe that in the defect prediction domain, the real-time-based data collection (temporal sampling) is essential for better prediction quality. The software projects have evolved chronologically and the prior information is a better predictor. Further, the random sampling may lose vital information and hence, it causes negative impact on the prediction models. The correlation value of the temporal sampling technique in defect prediction is small (0.21) and therefore, one could argue on the practical usage of this model. However, we provide a qualitative indication of the effect of temporal data on prediction quality. Hence we can support the hypothesis 1.1.
In the cost estimation, in contrast to the defect prediction domain, the time factor has no significant effect on predicting the final cost of a project. A good reason for this observation is that the cost estimation data has no information about the relationship between the time and the cost factors. Usually, a software project undergoes several iterations or revisions before it released for operations. Each of these revisions needs a certain amount of cost in terms of working hours, money or space (lines of codes). However, the explored dataset does not provide any information about this relationship, instead it provides final cost of projects with the year of completion. Therefore, we cannot support the hypothesis 1.2.
These findings encourage us to further investigate the influence of temporal features and the prediction models on defect prediction quality and it is investigated in the next chapter.

6.5 Threats to Validity

The general threats explained in the Section 3 are valid for this experiment too. Additionally, this experiment is affected by the following threats:
Project dependency: We use only two open-source projects Eclipse and Netbeans. Since the projects are in the same family – Integrated Development Environment (IDE) – and open-source, the code generation and bug reporting methods could be of similar style. Therefore, in order to generalize this finding we may want to investigate projects from different development environments.
Choice of algorithm: We use only linear regression model for this experiment. To avoid the algorithm dependency we may want to run the same experiment with non-linear regression models.
Choice of observation periods: For the above experiment, we collect data from only one observation period: March 01, 2004 - March 01, 2008. Though we provide good reasons for selecting this period we still violate the conditions for the generalizability of our findings. As we collect data from an ample time period we are confident that the out come will not be substantially change.
Finally, lack of data in cost estimation domain negatively affects the generalizibility of our findings. Indeed, everyone in the cost estimation research field encounters this drawback.

7

Impact of Temporal Features and Models on Prediction Quality [1]

One of the central questions in software engineering is how to write bug-free software. Given that it is virtually impossible to do so researchers are striving to develop approaches for predicting the location, number, and severity of future/hidden bugs. Such predictions can be used by software development managers to (among other things): (1) identify the most critical parts of a system that should be improved by respective restructuring, (2) try to limit the gravity of their impact by, e.g., avoiding the use of these parts, and/or (3) to plan testing efforts (parts with most defects should be tested most frequently). Mining Software Repositories (MSR) [2] is one such research community and, one of their main objectives is to develop novel approaches for predicting defects. Also, the foundation for this work was laid by the MSR challenge 2007 [3]. So, the challenge was to predict defects that will happen in Eclipse project on January 2007. Further, this work was encouraged by the outcome of our previous project. As we mentioned in the Introduction section, so far, several defect prediction models have been developed. But still no study has concluded on a best technique that always completely outperforms other techniques . Therefore more research is required to develop further techniques that will improve the prediction quality. In the following sections we discuss the main issues that we have to encounter and the approaches that we use.

7.1 Preliminaries

In our previous work we uncovered temporal information i.e. prior information is a better defect predictor. Further, we uncovered that randomly selected information has negative impact on prediction quality. Hence, taking lessons from the previous work, we use all prior information for this work. However, before mentioning the main issues that we are going to address in this project, we define two technical terms: `static features` and `temporal features`.

[1] Parts of this section were published in [Bernstein et al., 2007]
[2] http://2011.msrconf.org/
[3] http://msr.uwaterloo.ca/msr2007/

Definition: `Static features`
Static features are file-related features that are extracted over the entire observed period.

Definition: `Temporal features`
As static features, temporal features are file-related features but, contrast to the static features, temporal features are extracted over a temporal window (i.e. 1-month, 2-month etc.), describing the evolution of static features over time.

In summery, we encounter the following issues in this project:

1. Should we use temporal features or static feature to train prediction models?

2. Which learning algorithm (linear or non-linear) should be appropriate for training models since there are several such learning algorithms?

3. Can we convert the findings into actionable knowledge?

In this chapter we therefore introduce a step-by-step approach to address the above issues. Our intention is to compare the prediction performances between static and temporal features together with linear and non-linear models. Hence, we define the following hypotheses to counter our first research question:

H 2.1: *Models trained from temporal features are more predictive than models trained from static features.*

H 2.2: *Non-linear models are more precise on predicting defect than linear models.*

In the next section we discuss about the data, algorithms and the performance measures of those algorithms.

7.1.1 Data Description- CVS and Bug Reports

As we have mentioned already, this work was encouraged by MSR 2007 challenge. They provided Eclipse project data: CVS and bug reports for analysis. We then link CVS data with bug reports and construct features as mentioned in the Section 2. For this project we use six plugins of Eclipse project: updateui, updatecore, search, pdeui, pdebuild, and compare. We consider only unique file names and source code file types *.java. Further, we consider only the files that were not marked as dead within the observation period. Table 7.1 lists the components, release dates and the number of files contained in each component.

We omit 59 files out of 3890 as they do not have a sufficient number releases to provide temporal information for our experiment. Another example for exclusion is files with modification reports that contain wrong or unavailable release dates.

Plugin	First Release	Last Release	#Files
updateui	Jan 03, 2001	Jan 18, 2007	757
updatecore	Jan 03, 2001	Jan 18, 2007	459
search	May 02, 2001	Jan 30, 2007	540
pdeui	Mar 26, 2001	Jan 30, 2007	1621
pdebuild	Dec 11, 2001	Jan 12, 2007	198
compare	May 02, 2001	Jan 30, 2007	315
Total			3890

Table 7.1: Investigated components and their released dates

The composition of a dataset for training models is same as described in the Section 5. However, the formal description of the dataset is slightly different than in the previous section. Hence, we first provide the formal definition of the dataset as below:

Assume that the observed period is d months. $Y_T = \{y_{T,1}, y_{T,2}, ..., y_{T,j}, ..., y_{T,s}\}$ is a vector of dimension s (s is the number of observed files) and $y_{T,j}$ is the number of bugs reported for file j at time T, where T is $1 < T \leq d$ (in this case T is December 2006). if $X_d = \{f_{d,1}, f_{d,2}, , f_{d,i}, ..., f_{d,n}\}$ is a feature vector of dimension n and $f_{d,i}$ is a file feature i computed during the observation period (d), where $n \in N$ and $s >>> n$, then constructed dataset is given by $\bigcup_{t=d}^{T-1} X_t, Y_T$. This dataset is used to train models for predicting defects in time $T + 1$.

7.1.2 The Data: Features

For each of the investigated 3831 source files we use the CVS and bug information of the above six plugins to compute the features listed in Table 7.2. The feature computation is as mentioned in the Section 2. Most of the features listed in the Table 7.2 are self-explanatory. However, some of the features need additional context and provided below:

`RevisionAuthor` represents the workload of an author per file. We first count the number of revisions for a file during the training period and divide this number by the number of unique authors that edited the file.

Feature `AlterType`, classifies each modification into large, medium, and small, according its size relative to the lines of code modified in the source files. If the sum of lines added and deleted is more than double of the current code length then `AlterType` of this modification is large. If the modification relative to the code length is between 1 and 2 then `Altertype` is medium. If the size of the change is below 1 then `AlterType` is small. Indeed, this feature quantifies the the amount of change done by authors.

`RevisionAge` denotes how often a file is changed for adding new features or fixing bugs during its lifetime. The age of a file is measured using number of months and it is counted from the file's first commit date to December 2006. The number of revisions of a file during the training period is divided by the age of the file will generate the feature `RevisionAge`.

Features 7–18 contain temporal file features. Essentially, those features are computed during

#	Name	Description
	Features from Source code	
1	LOC	Number of lines of codes
	Features from Versioning System	
2	Releases	Total number of releases
3	RevisionAuthor	Number of revisions per author
4	AlterType	Amount of modification done relative to LOC
5	AgeMonths	Age of a file in months
6	RevisionAge	Number of revisions relative to the age of a file in months
	Temporal Features	
7	Revision1Month	Number of revisions of a file from Dec 1 to 31 of 2006
8	DefectAppearance1Month	Number of releases of a file with defects from Dec 1 to 31 of 2006
9	ReportedI1Month	Number of reported problems of a file from Dec 1 to 31 of 2006
10	Revision2Months	Number of revisions of a file from Nov 1 to Dec 31 of 2006
11	DefectAppearance2Months	Number of releases of a file with defects from Nov 1 to Dec 31 of 2006
12	ReportedI2Months	Number of reported problems of a file from Nov 1 to Dec 31 of 2006
13	Revision3Months	Number of revisions of a file from Oct 1 to Dec 31 of 2006
14	DefectAppearance3Months	Number of releases of a file with defects from Oct 1 to Dec 31 of 2006
15	ReportedI3Month	Number of reported problems of a file from Oct 1 to Dec 31 of 2006
16	Revision5Months	Number of revisions of a file from Aug 1 to Dec 31 of 2006
17	DefectAppearance5Months	Number of releases of a file with defects from Aug 1 to Dec 31 of 2006
18	ReportedI5Month	Number of reported problems from Aug 1 to Dec 31 of 2006
	Features from both Bug tracker and Versioning system	
19	DefectReleases	Number of releases of a files with defects relative to total number of releases
20	LineAddedIRLAdd	Number of lines added to fix a bug relative to total number of lines added
21	LineDeletedIRLDel	Number of lines deleted to fix a bug relative to total number of line deleted
22	ReportedIssues	Total number of reported problems
	Target Features	
23	hasBug (Target)	Indicates the existence of a bug in a file
24	hasNumberBug (Target)	Indicates the number of bug in a file

Table 7.2: Extracted features (variables) from CVS and bug reports

different temporal windows (1, 2, 3, and 5 months) backwards from the December 2006 releases. The other features are static features and they are computed during training periods (backwards from December 2006) of the components.

Lines of codes are added/deleted both when fixing a bug and when adding new features. However, they need to be separately implemented and thus we introduce two features `LineAddedIRLAdd` and `LineDeletedIRLDel` that represent the number of lines added/deleted to fix a bug relative to the total number of lines added/deleted.

Feature `ReportedIssues` describes how many issues have been reported during the training period. Here, we count issues as the reports categorized as blocker, critical, major, minor, normal and trivial in bug tracking system. We determine the reported date of a bug as described in the Section 2.

Target feature: `hasNumberBug` is a target feature that recodes the number of bugs, which have been reported for each file. The other target feature is `hasBug`, which locates the buggy files. Both target features are computed during the labeling period. The bug reported date is determined as described in the Section 2.

The full description of the features in the Table 7.2 can be found in the Section 5.

7.1.3 Choice of Algorithm and Performance Measurements

One of our objectives of this experiment is to compare the defect prediction quality between linear and non-linear models. To that end we use the WEKA data mining framework [Witten and Frank, 2005] to implement linear regression models, J48 decision tree learner: a re-implementation of C4.5 [Quinlan, 1993] and linear regression trees (M5P) [Wang and Witten, 1997].

The linear regression model and the linear regression trees are trained to predict the number of bugs that will be reported for each file during the month January 2007. We measure prediction quality of the regression models using Spearmans Rank correlation (ρ), root mean squared error (RMSE), and mean absolute error (MAE). The decision tree model (J48) predicts the location of the bugs, i.e which file is going to buggy on January 2007. For the location prediction we learned the decision tree model, which computes the probability distribution over the two possible classes: `hasBug` and `hasNoBug`. Since decision trees are usually used to predict classes, we picked the class with the highest probability and computed the confusion matrix of the model, which can (partially) summarize the models accuracy. The problem of the accuracy as a measure is that it does not relate the prediction of the prior probability of the classes. This is especially problematic in heavily skewed distributions such as the one we have. Therefore, we also used the receiver operating characteristics (ROC) and the area under the ROC curve, which relate the true-positive rate to the false-positive rate resulting in a measure uninfluenced of the prior (or distribution) [Provost and Fawcett, 2001] [Witten and Frank, 2005].

The advantage of selecting these three models is that all of them are simple and human readable in contrast to Support Vector Machine (SVM) [Cortes and Vapnik, 1995] or Artificial Neural Network (ANN). Indeed, readability is really important to explain the possible causes for defects.

However, the decision tree and the regression tree are more computationally expensive than the linear regression model.

7.2 Method Implementation

In this section we implement experiments to counter the issues by validating our hypotheses mentioned in the Section 7.1. First, we predict the location of bugs that will happen on January 2007 (target period) to test the predictive power of the features listed in the Table 7.2. Second, we predict the number of bugs reported for each file on the same target period. Third, we compare the prediction quality between temporal and static features together with linear models and non-linear models. Finally, we apply the findings into actionable event.

7.2.1 Locate Buggy files : Is our feature list powerful enough?

The objective of this experiment is to locate source code files with defects that will happen on January 2007. To that end we first prepare a dataset as described in Section 7.1.1 to train prediction models. We use J48 decision tree learner implemented in WEKA data mining tool. To test our first proposition – that temporal features would improve the prediction quality – we train the models from data collected backwards from December 2006. The models are trained with different base-sets of features either using static features whatsoever (i.e., excluding features 7-18 of Table 7.2) or using the temporal features for different window sizes of 1, 2, 3, and 5 months (i.e., choosing a selection of features 7-18 representing the window size under investigation). Since choosing a good feature set for the prediction model is imperative for a good prediction performance we used a number of wrapper-based feature selection methods such as sequential forward selection [Kohavi and John, 1997]. These methods compare the prediction performance of different subsets of the features within the training set to find the best performing subset. The best performing subset of features was then used to induce the prediction model, which was then tested on the test set. To test the models' prediction quality we use a separate test set collected on January 2007. This procedure ensures that only information available on December 31, 2006 was used to predict the location of defects or the number of bugs in January of 2007.

Result and Discussion

Table 7.3 summarizes the results of these experiments. It shows the list of features chosen by the feature selection method, the accuracy, and the area under the ROC curve of the prediction for each of the trained models. It is interesting to observe that the only feature chosen for all models is the `LineAddedIRLAdd` (feature # 20 in the Table 7.2), which relates the numbers of lines added due to bug fixing to the number of lines added due to adding new features. Even though this feature is chosen by all models it does not seem to play a pivotal role in the models, as it does not show in none of the trees first two levels. Another interesting observation is the dominance of the

temporal features (features 7-18): not only do they get chosen whenever possible, they also show up at the root of the tree (see column 3) whenever available. When looking at the target performance measures accuracy and area under the ROC curve (AUC) we clearly see the dominance of the prediction that can take advantage of temporal features compared to the one that cannot. In terms of accuracy, we can clearly see that temporal information boosts the performance, but that more recent temporal data is more useful than older one. We can, hence, hypothesize that modules with bugs are likely to have bugs in later versions, but that over longer periods of time those bugs could be fixed. In other words, bugs are likely to survive some versions, but are fixed after some. The same fact was uncovered by several researches [Hassan and Holt, 2005],[Kim et al., 2007], but using different techniques.

Method	**Features**	**Root Node**	**Accuracy**	**AUC**
Static features	2,3,4,6,19,20,21,22	`DefectReleases`	96.5805%	0.8611
1-month	3,7,9,20,22	`ReportedI1Month`	99.1125%	0.8948
2-months	3,4,10,11,12,20,22	`ReportedI2Month`	98.8776 %	0.8933
3-months	2,3,4,6,13,14,15,19,20,21,22	`DefectAppearance3Months`	98.6427%	0.9039
5-months	2,3,4,5,6,16,18,19,20,21,22	`ReportedI5Months`	97.7813%	0.8663
Significant	7,8,10,15,18,20,21	`DefectAppearance1Month`	99.1647%	0.9251

Table 7.3: Results of different models for defect location prediction (Accuracy of default strategy 96.35%)

The difference between 96.58% (static features) and 99.16% (significant features) in accuracy does not look significant enough to warrant the computation of the temporal measures. Note, however, that the sole use of accuracies is misleading since they are heavily dependent on the prior distribution of the data. In our case, where the class distribution is highly skewed (we have 140 buggy classes versus 3691 non-buggy ones), it is simple to attain a high accuracy: "just" assigning "non-buggy" to every file (the default strategy) one gets an accuracy of 96.35% ($= \frac{3691}{3691+140}$) according to the confusion matrix for the best model (including significant features) as shown in Table 7.4. Hence, the use of accuracy as a measure for the quality of the prediction is misleading. We, therefore, computed the receiver operating characteristics (ROC) for each of the methods and the area under the ROC-curve (AUC), both of which provide a prior-independent approach for comparing the quality of a predictor [Provost and Fawcett, 2001].
Figure 7.1 graphs the ROC curves for all the chosen methods. The x-axis shows the false-positive rate and the y-axis, the true positive rate. Note that a random bug assignment is also shown as a line from the origin $(0, 0)$ to $(1, 1)$ and that the ideal ROC curve would be going from the origin straight up to $(0, 1)$ and then to $(1, 1)$. The Figure clearly shows that all prediction methods provide a significant lift in predictive quality over the random assignment. But the methods have very interesting differences in terms of quality. Since one method dominates another when its ROC-curve is closer towards the upper left corner, we can see how the non-temporal prediction model is dominated along almost the whole frontier by the temporal models. The figure also shows how the method using significant features dominates the other methods along almost the

whole frontier whilst employing fewer features (see Table 7.3). Further note that the dominance of the ROC-curve is reflected by a larger area under the ROC curve (AUC) as listed in Table 7.3.

	predicted buggy	predicted bug free
has bugs	117	23
has no bugs	9	3682

Table 7.4: Confusion Matrix for the significant features model

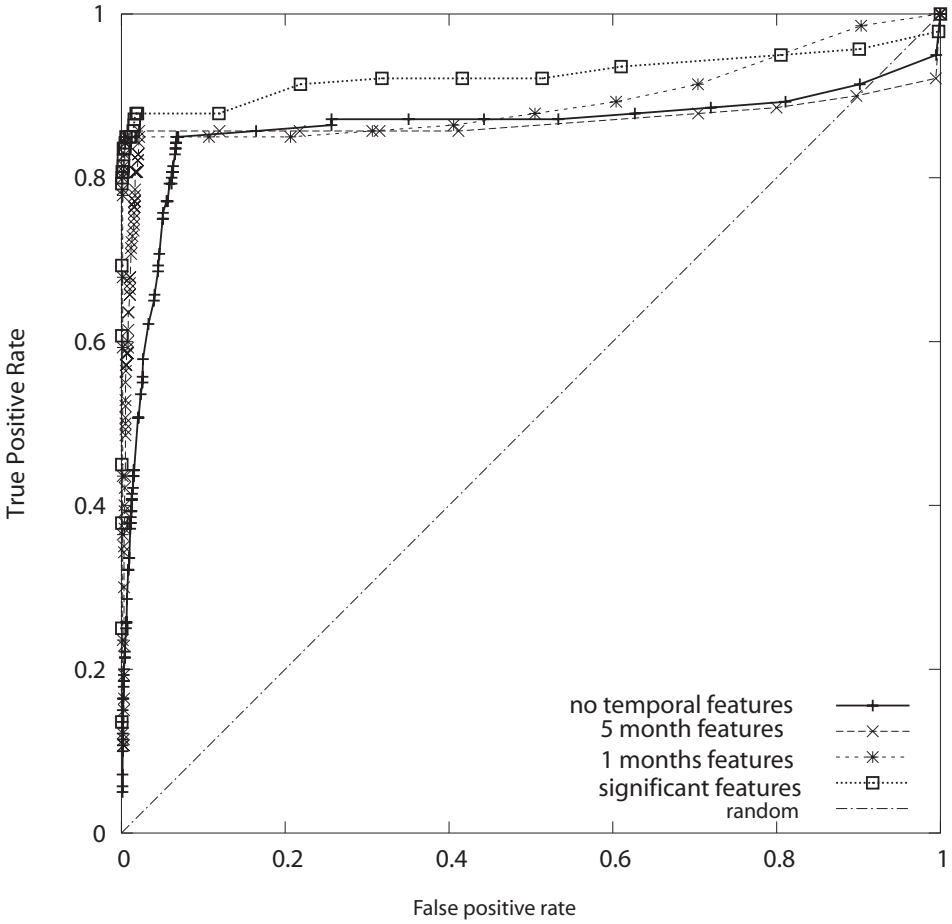


Figure 7.1: ROC-curves of defect prediction methods.

To further improve our understanding of the structure of the prediction methods, we succinctly compare the two top levels of the prediction trees for the static, the 1-month, and the significant feature model. As the top levels of the trees depicted in Figures 7.2, 7.3 and 7.4 show, even the model without temporal features (see Figure 7.2) heavily relies on the quasi temporal feature `DefectReleases`, which computes the fraction of past releases with bugs. The next most important static feature seems to be `LineDeletedIRLDel` signifying the importance to distinguish

between changes due to bug fixing versus changes due to the addition of new features. Note that this seems to be a very important distinction, as the related `LineAddedIRLAdd` feature is the most important static feature in the tree (see Figure 7.3).

```
DefectReleases <= 5.263158: NO (3423.0/11.0)
DefectReleases > 5.263158
|   DefectReleases <= 21.95122
|   |   LineDeletedIRLDEl <= 17.518248: NO (222.67/35.0)
|   |   LineDeletedIRLDEl > 17.518248
|   |   |   Releases <= 206: NO (34.33/10.0)
|   |   |   Releases > 206
|   |   |   |   RevisionAuthor <= 3.125: YES (7.0)
|   |   |   |   RevisionAuthor > 3.125: NO (3.0/1.0)
|   DefectReleases > 21.95122
|   |   DefectReleases <= 67.142857
|   |   |   AlterType = large
```

Figure 7.2: Static features

```
ReportedIssues1Month <= 0: NO (3692.0/21.0)
ReportedIssues1Month > 0
|   Revision1Month <= 1: YES (105.0/2.0)
|   Revision1Month > 1
|   |   LineAddedIRLADD <= 3.636364: NO (16. 0/2.0)
|   |   LineAddedIRLADD > 3.636364
|   |   |   ReportedIssues <= 7: YES (15.0/1.0)
|   |   |   ReportedIssues > 7: NO (3.0)
```

Figure 7.3: 1-Month temporal features

```
DefectAppearance1Month <= 0
|   ReportedI5months <= 0: NO (3599.0/9.0)
|   ReportedI5months > 0
|   |   Revision2Months <= 4: NO (86.0/7.0)
|   |   Revision2Months > 4
|   |   |   LineAddedIRLADD <= 1.359223: NO (2.0)
|   |   |   LineAddedIRLADD > 1.359223: YES (5.0)
DefectAppearance1Month > 0
|   Revision1Month <= 1: YES (105.0/2.0)
|   Revision1Month > 1
```

Figure 7.4: significant features

As concluding remarks from this experiment, we can say that the experiment for defect location prediction clearly shows that one *can, indeed, predict the location of bugs with a high accuracy*. We can also say that this accuracy is based (to a large extent) on temporal features. We hypothesize that one reason for the effectiveness of temporal values is that bugs the usually survive more than one release. Other reasons might be the fact that complicated/complex or badly engineered classes are likely to exhibit bugs repeatedly unless they are re-engineered. Furthermore, we ob-

Model	Features	Root Node	ρ	MAE	RMSE
Static features	2,3,5,21,22	`LineDeletedIRLDel`	0.863	0.0524	0.1898
1-months	2,3,5,6,7,8,9,19,20,21	`Revision1Month`	0.941	0.0226	0.1272
2-months	2,3,5,6,10,11,12,19,21	`Revision2Months`	0.950	0.0249	0.133
3-months	2,5,13,14,15,19,20	`Revision3Months`	0.966	0.0241	0.1298
5-months	3,5,16,17,18,20,21	`Revision5Months`	0.942	0.0326	0.1575
Significant	2,3,5,7,8,11,13,14,15,19	`Revision1Month`	0.963	0.0194	0.1119

Table 7.5: Results of different models for defect location prediction with M5P

serve that the most important static features for prediction are the relations between line changes due to feature additions versus line changes due to bug fixing in the past – a type of feature not yet largely investigated in the literature, which clearly deserves more attention.

In this experiment we show that our feature set is powerful enough for locating defects, but, developers further interest on understanding number of bugs per file in advance, since they can plan for fixing effort. Hence, we are curious about whether our feature list is clever enough for predicting the number of bugs per file. In the next experiment we counter this issue.

7.2.2 Predicting the Number of Bugs: Is our feature list powerful enough?

The goal of the second group of experiments is to establish if our approach can amply predict the number of bugs that occur in any given file. This task is more difficult than the last, as it not only has to predict the existence of bugs (*i.e.* , if $\#bugs > 0$) but the actual numbers of bugs. Since we believe that the task of predicting the number of bugs exhibits non-linear properties (a belief, for which we show evidence in Section 7.2.3) we decided to use a non-linear regression approach. To preserve the comprehensibility of the model as well as the comparability of the model to the defect location prediction above, we chose the Weka implementation of the M5P tree regression algorithm [Wang and Witten, 1997]. A regression tree model combines a decision tree an a linear regression by partitioning the feature space with a decision tree and then provides a linear regression equation for each of the tree's leaves. The model can, thus, predict a number by assigning any instance (i.e., entity to predict) to a leaf and then performing the associated regression to compute a number. This approach has the advantage that the regressions at the leafs do not have to be linearly connected – the tree provides the non-linear partition, the linear regressions predict the number.

The predictive power of temporal features. To test our proposition – that temporal features improve the prediction quality – we followed the same procedure as above: we train the model with different base-sets of features either using static features whatsoever (i.e., excluding features 7-18 of Table 7.2) or using the temporal features for different window sizes of 1, 2, 3, and 5 months backwards from the December 2006 releases. The models are tested with the same set of features

extracted on January 2007.

Results and Discussion

Table 7.5 summarizes the results for this comparison. Like in Table 7.3 it lists the name of the model, the features chosen by the feature selection algorithm and the root node of the regression tree. As performance measures, it lists the Spearman's correlation (ρ) between the prediction and the actual data, the mean absolute error (MEA), and the root mean square error (RMSE). The results mostly mirror the ones form the location prediction experiments.

The models that can rely on the temporal features do so and even use it as the main feature for the decision tree. In contrast to the the location prediction, though, the root nodes of the tree do not have the most recent available number of reported issues or bugs (*i.e.* , `ReportedI1Month`, `ReportedI2Month`, `ReportedI2Month`, `DefectAppearance1-Months`, or alternatively `DefectAppearance3Months`), but exclusively uses the number of available (*i.e.* , `RevisionXMonth`, where `X` is the most recent available number for learning). While this is surprising at the surface, further investigation clarifies the issue: when investigating the features chosen by the feature selection algorithm we can clearly see that the elements chosen as root nodes in the defect location prediction are used in the defect number prediction. In contrast to the defect location prediction, they are not at the root of the partitioning decision tree but are mostly used in the regression function at the leafs. Consider, for example, the model induced for significant-features model as shown in Figure 7.5. At the top we can clearly see the decision tree that partitions the feature space, using only some of the features. Below, the figure shows the first of 8 linear regression models. This particular model is called if the rule at the root of the tree (`Revision1Month` $\leqslant 0.5$) is true. As the regression shows it uses the root node of the defect prediction decision tree `DefectAppearance1Month` with the second strongest weight in the regression.

Similar to the bug prediction case Table 7.5 also clearly shows how the models with temporal features dominate the model without them. The difference in the Spearman's ρ (0.963 for temporal features vs. 0.863 without temporal features) is striking. The error rates MAE and RMSE mirror this behavior. Therefore, the results support our argument that the temporal data improve the accuracy of prediction model.

Exploring the prediction error. A closer look at the error rates in Table 7.5 also reveals that the RMSE is an order of magnitude larger than the MAE for all the models. This indicates that there are some large errors, which weigh in more heavily in the RMSE. Table 7.6 shows the histogram analysis of residual error of the significant-features model. As the table shows the bulk of the prediction has no (74.07%) or little (i.e., error $\leqslant 0.5$; in 98.69%). Nonetheless, a few predictions exhibit an error larger than 1. It is these predictions that mostly influence the error. When removing the file with a prediction error of 2.93, the MAE is lowered to 0.0194, but the RMSE is lowered to 0.0014, a full order of magnitude smaller. It is, hence, this one outlier that mostly contributes

```
Revision1Month <= 0.5 : LM1 (1348/0%)
Revision1Month >  0.5 :
|   LineAddedIRLADD <= 0.098 :
|   |   AgeMonths <= 33.667 :
|   |   |   Releases <= 65.5 : LM2 (343/0%)
|   |   |   Releases >  65.5 :
|   |   |   |   AgeMonths <= 15.95 : LM3 (112/87.619%)
|   |   |   |   AgeMonths >  15.95 : LM4 (266/26.955%)
|   |   AgeMonths >  33.667 : LM5 (975/0%)
|   LineAddedIRLADD >  0.098 :
|   |   Defectappearance3Months <= 0.5 : LM6 (619/42.644%)
|   |   Defectappearance3Months >  0.5 :
|   |   |   Revison3Months <= 1.5 : LM7 (81/171.567%)
|   |   |   Revison3Months >  1.5 : LM8 (87/210.532%)

LM num: 1
NumberofErroresLastMonth =
        0 * LineAddedIRLADD
        + 0 * AgeMonths
        + 0.0005 * Revison3Months
        - 0.0005 * Defectappearance3Months
        - 0.0013 * ReportedI3Months
        - 0 * Releases
        + 0 * RevisionAuthor
        - 0.0002 * Revision5Months
        + 0.0002 * DefectAppearance5Months
        - 0.0002 * Revision1Month
        + 0.0019 * DefectAppearance1Month
        + 0.0043 * ReportedI2Months
        - 0.0003
```

Figure 7.5: Excerpt of bug prediction model relying on significant features

to the RSME. Furthermore, when removing all 5 files with an error larger than 1 we get a MAE of 0.0177 and a RMSE of 0.0095. We can, thus, conclude that the prediction error of our method is, in general, very small.

Error Interval	Frequency	Absolute	Cumulative
0	2838	74.08 %	74.08 %
$0 < e \leqslant 0.5$	943	24.61 %	98.69 %
$0.5 < e \leqslant 1$	45	1.17 %	99.87 %
$1 < e \leqslant 1.5$	4	0.10 %	99.97 %
$1.5 < e \leqslant 2$	0	0 %	99.97 %
$2 < e \leqslant 2.5$	0	0 %	99.97 %
$2.5 < e \leqslant 3$	1	0.03 %	100.00 %

Table 7.6: Residual error histogram for significant-feature model

As concluding remark from this experiment, we can say that our non-linear bug prediction model supports our proposition that temporal features are imperative for an accurate prediction – without them the Spearman's rank correlation ρ between the predictions and the actual error numbers is lowered from 0.963 to 0.863. Second, we can clearly see how our model is highly accurate for most predictions and that most of the residual errors are introduced by 5 predictions of 3831. However, we want to compare our prediction models with some other prediction models that use the same dataset. Hence, in the next section we discuss this task.

Comparison with other defect predictions using the same data set The MSR Mining Challenge

2007,[4] which provided the data we used for our study, had a similar task as its Challenge #2. The main difference between our approach and the challenge task is that we chose to make our predictions on the file level and the Challenge task required participants to predict the number of bugs for 32 plug-ins[5] (i.e., summarizing the bugs for all their classes). Two methods, one in two versions were submitted to the mining challenge. `C-ESSEN` by Adrian Schröter [Schroter, 2007] predicted the bugs based on the import statements used in the files. This is a measure we did not use at all. ULAR by Joshi *et al.* [Joshi et al., 2007] uses features computed in the last month to make a prediction for next month. A second version of ULAR extends those predictions with a trend analysis. Last, an ad-hoc method used as an comparison by Thomas Zimmermann (called 1 Year ago) simply takes the measures from 2006 to make the prediction for 2007. Table 7.7 shows the Spearman's rank correlation (ρ) for all the methods as well as our significant-feature model. The results show that our approach is better at ranking the files according to their expected bugs. The ranking, rather than the precise prediction of the number of bugs, is actually an important task when one tries to make an optimal assignment of resources (*i.e.* , programmers) to tasks (*i.e.* , the fixing of bugs) [Provost and Fawcett, 2001]. Note, however, that the other models are making their prediction on 32 modules whereas we limit ourselves to only 6, which is a much simpler task.

Model	n	ρ
C-ESSEN (imports) [Schroter, 2007]	32	0.67
ULAR (Last month + trends) [Joshi et al., 2007]	32	0.81
ULAR (Last months) [Joshi et al., 2007]	32	0.84
1 Year Ago	32	0.91
significant-features model	6	1.00

Table 7.7: Spearman's ρ for MSR Mining Challenge 2007 results, where n is the number of components. Our approach is significant compared to the others at $\alpha = 0.1$.

Summarizing, we compare the performance of our approach to similar tasks (as we did not find any work on the same task). We find that our approach exhibits a better performance compared to others with respect to the Spearman's rank correlation. Hence, we can can guarantee our models for applications.

In the above experiments we validate our first hypothesis (H 2.1): *Models trained from temporal features are more predictive than models trained from static features*. In the next section we discuss step-by-step approach to address the second hypothesis.

7.2.3 Are Non-Linear Models Better than Linear Models?

We conduct this experiment to counter the second hypothesis (H 2.2): *Non-linear models provide a superior prediction quality than the simple linear regression models*. Specifically, we stated that the

[4] http://msr.uwaterloo.ca/msr2007/challenge/`http://msr.uwaterloo.ca/msr2007/challenge/`

[5] http://msr.uwaterloo.ca/msr2007/challenge/plugins.txt`http://msr.uwaterloo.ca/msr2007/challenge/plugins.txt`

	LM			M5P		
Model	ρ	**MAE**	**RMSE**	ρ	**MAE**	**RMSE**
Without temporal features	0.844	0.0569	0.1902	0.863	0.0524	0.1898
1-Month	0.935	0.0306	0.1311	0.941	0.0226	0.1272
2-Months	0.919	0.039	0.1421	0.950	0.0249	0.133
3-Months	0.891	0.0471	0.1523	0.966	0.0241	0.1298
5-Months	0.918	0.0423	0.1611	0.942	0.0326	0.1575
Significant Features	0.929	0.0319	0.1227	0.963	0.0194	0.1119

Table 7.8: Comparison of linear model (LM) and Non-linear model (M5P), ρ is the Spearman's rank correlation.

non-linear models are able to exploit the non-linear relationships between the features to make more accurate number of bugs predictions. By non-linear we mean, a relationship that cannot be captured by a weighted sum of simple, continuous functions of the single features (as done by a linear regression), but may require functions of two or more features. To explore this hypothesis we re-ran experiments outlined in sub-section 7.2.2 with a standard linear regression algorithm.

Results and Discussion

Table 7.8 shows the results of this analysis comparing the Spearman's rank correlation (ρ), the mean absolute error (MAE), and the root mean squared error (RSME) for the linear model (LM) – a standard linear regression – and the non-linear model in the form of the M5P algorithm. The results show that the non-linear significantly outperforms the linear model for all performance measures (results for pairwise t-test significant at: $p = 1.09\%$ for ρ, $p = 0.29\%$ for MEA, and $p = 2.42\%$ for RMSE). The dominance is, however, not constant. For the data sets without temporal features the LM and M5P have a very similar performance. The more recent temporal features the more pronounced is the dominance of the non-linear model. This would lead us to hypothesize that the static features exhibit a non-linear relationship to the number of bugs. If we explore the actual model this hypothesis is confirmed. Consider again Figure 7.5, which shows the bug prediction model for significant-features. As the model clearly shows the temporal features are heavily used within the non-linear element of the model: the decision tree that partitions the feature space. Nonetheless, the temporal features are also reused in the linear part of the model: the leaf-based regressions.

We can, thus, conclude that (1) the temporal features have both linear and non-linear elements with respect to the number of bugs and (2) that the M5P's capability to exploit both linear and non-linear elements clearly results in more accurate results.

7.2.4 Turning Findings into Actionable Events

So far we have shown that our prediction models are superior in predicting defects in terms of location and number per file. However, can we use these models to identify the most Suspicious Eclipse plugins on January 2007? To that end we apply the best performing prediction model to

Pugin	Actual bugs	Predicted bugs	Accuracy
pdeui	83	68.8999	83.0119%
compare	36	29.5561	82.1002%
pdebuild	20	16.7421	83.7106%
updateui	10	8.6371	86.3718%
updatecore	8	7.1928	89.9104%
search	1	1.0663	93.7836%

Table 7.9: Predicted and actual number of bugs for the six Eclipse plugins in January 2007.

identify the most critical Eclipse plugins (out of the six). These plugins need to be considered first when planning refactoring and testing efforts. By 'critical' we mean plugins for which our model predicts the highest number of bugs for January 2007. Table 7.9 lists the results with the actual number of bugs, the predicted number of bugs, and the accuracy of the prediction model.

From a managers point of view the number of predicted bugs clearly indicates that refactoring as well as testing effort needs to be dedicated to the two plugins `pdeui` and `compare`. In particular, `pdeui` is indicated as a critical plugin that, according to our model, will be affected by around 69 bugs in January 2007. This mirrors the actual number of bugs, which was 83. On that note we conclude that such predictions provide a valuable input for software project managers to plan refactorings and tests.

7.3 Concluding Discussion

In this chapter we uncover some techniques to improve defect prediction quality. First, we disclose that the temporal features, contrast to static features, improve the prediction quality. Indeed temporal features describe the evolution of static features over time. Moreover, they contain latest historical information that is more useful in predicting defects. Next, we reveal that the non-linear models are superior to linear models in predicting defects. This implies that the relationship between features and defects is not linear and complex. Hence, we support hypotheses 2.1 and 2.2. Since we uncover an interesting aspect on defect prediction domain, we compiled this finding into a publication [Bernstein et al., 2007] and presented in the workshop IWPSE 2007[6]

7.4 Threats to Validity

The threats explained in the section 3 are valid for this experiment too. In addition to those there are some other threats particular to this chapter.

Project dependency: We choose only an Eclipse data set that represents only one project-family. While we followed good data mining practices to ensure the generalizability of our findings, the

[6]http://iwpse2007.inf.unisi.ch/

data might behave Eclipse idiosyncratic such as a common culture of bug-reporting or code documentation/fixing, programming language dependencies (Eclipse only uses Java), etc.
Time dependency: We only looked at the prediction quality for the last month (January 2007) of the data set. We intend to ascertain the generalizability of our findings by exploring the quality of the prediction for other months within the Eclipse data set and for other projects altogether.
Feature and algorithm selection: We only used one off-the-shelf feature selection and non-linear induction algorithm. It might, therefore, be that the resulting feature set and the model are sub-optimal. Following good data analysis practices we should try a whole set of algorithms to determine the most predictive model - a task that we will undertake in the near future. Nonetheless, we are confident that the use of other algorithms will not substantially change our findings. Moreover, we expect them to potentially make them even more pronounced than at present.
Further, our candidate features were chosen by our study of the literature and some of our own thoughts regarding temporal features. In order to ensure an optimal performance of the resulting models, we need to explore the full space of possibly applicable measures (or features) reported in the literature. We hope to expand the feature space in the next chapter. Similar to the feature selection, however, we think that such an exploration would make our finding more pronounced without canging the inferred conclusions. Last and most importantly, our attempt could be seen as a post-prediction rather than a pure prediction.

Part V

Time and Prediction Quality

8

Defect Prediction Varies Over Continuous Time Periods[1]

Today, software bugs remain a constant and costly fixture of industrial and open source software development. To manage the flow of bugs, software projects carefully control their changes, using software configuration management (SCM) systems (Such as CVS or SVN), capturing bug reports using bug tracking software (such as Bugzilla, IssueZilla or Jira), and then recording which change in the SCM system fixes a specific bug in the change tracking system. On the other hand, mining software repository community utilizes this information to develop quality prediction models. These models are valuable tools for software developers so that the developers can identify suspicious modules in advance. Hence, the reliability of those prediction models is crucial for software managers. A wrong prediction of the number or the location of bug may be costly. As we already mentioned in the introduction section, majority of those models were tested only in one time period or in a very few periods. Further, those models were trained using data collected from a long history. Such evaluation and training implicitly assumes that the evolution of a project and its underlying data distribution are relatively stable over time. Is this assumption always true? As in the testing period, does the prediction quality remain constant in other prediction periods? Those are interesting issues but, to our knowledge, those issues have not been investigated in the literature. In the next sections we discuss these issues in more details and develop an approach to address them.

8.1 Preliminaries

It is a well known fact that software systems evolve over time. Evolution has happened in terms of size, functions, developers etc. of the software systems. There are many examples for such software projects, such as Eclipse, Netbeans, Mozilla *etc.* Hence, software systems can be considered as evolving systems. Many interesting findings can be found in literature regarding the evolving systems. Among them, Tsymbal [Tsymbal, 2004] uncovered that the evolution rules of an evolving system are not always stable due to the change in its underlying data distribution. He

[1]Parts of this section were published in [Ekanayake et al., 2011] and [Ekanayake et al., 2009]. The publication [Ekanayake et al., 2011] is not contained into Appendix due to copyright issue.

calls this problem as concept drift. Further, Widmer *et al.* [Widmer and Kubat, 1993] discovered from daily experience that the meaning of many concepts heavily depends on implicit context. Changes in that context can cause radical changes in the concept. Further, they revealed that the models trained with old concepts may not comply with new concepts and that frequently update is necessary. Taking these facts into consideration, it becomes clear that the assumption *evolution of software systems is consistent over time* is not necessarily valid. Due to this fact our 3rd research question "*Is a prediction model reliable in predicting defects at every time period?*" and its sub questions arise. In this chapter, we define a step-by-step approach to address these issues. To that end we first define the following hypotheses:

H 3.1: *Defect prediction quality varies over time.*

H 3.2: *There exists periods of stability and change in prediction quality.*

H 3.3: *There exists features that influence for prediction quality.*

In the next section we define the data and algorithms that we use to validate our hypotheses.

8.1.1 Data Description- Eclipse, Netbeans, Mozilla and Open Office

We explore four open source projects; Eclipse, Netbeans, Mozilla and Open Office for this investigation. We explained in the Section 2 the reason behind selecting these four projects. As in the previous chapters we consider only unique file names and source code file types such as *.java in Eclipse and Netbeans,*.cpp in Mozilla, as well as *.hxx and *.cxx in Open Office during the observed periods of each project. Further, we consider only the files that were not marked as dead within the observation period. All data is collected from the projects Concurrent Versioning Systems (CVS) and Bugzilla as mentioned in the Section 2.

Table 8.1 shows an overview of the observation periods and the number of files considered. Moreover, Tables 8.2–8.4 provide detailed descriptions about components and the number of files of those components. In Eclipse, we consider the core components of the products Equinox, JDT, PDE and Platform available in June 2007. We select all the components from Netbeans and Mozilla available in June 2007 and February 2008 respectively. For Open Office we only use files from the SW component. This component relates to the product writer being the word processor of the OpenOffice suite.

8.1.2 The Data: Features

Table 8.5 lists the features that we use to train models. A detail description of each of these features can be found in Section 5. However, we provide a brief description of two features `chanceBug` and `chanceRevision` because their names are not self-explanatory.

Project	First Release	Last Release	#Files
Eclipse	2001-01-31	2007-06-30	9948
Mozilla	2001-01-31	2008-02-29	1896
Netbeans	2001-01-31	2007-06-30	38301
Open Office	2001-01-31	2008-04-30	1847
Total			51,992

Table 8.1: Analyzed projects: time spans and number of files. Note: As starting date we picked the first date at which all projects were under development (i.e. Jan 01)

Component	#Files	Component	#Files
ant_core	36	pde_build	20
ant_ui	294	pde_ui	430
apt_core	161	pluggable_core	10
apt_tests	121	pluggable_tests	5
apt_ui	11	search	126
cknaus	6	text_tests	150
compare	160	ui_home	292
equinox_incubator	5770	update_core	45
jdt_debug	435	update_home	7
jdt_ui	1864	update_ui	5

Table 8.2: Eclipse: Investigated components and number of files

The `chanceRevision` and `chanceBug` features describe the probability of having a revision and a bug in the future as used in the award winning *BugCache* approach ([Kim et al., 2007]) discussed in Section 2. We compute those two features using the formula $1/2^i$, where i represents how far back (in months) the latest revision or bug occurred from the prediction time period. If the latest revision or bug occurrence is far from the prediction time period, then i is large and the overall probability of having a bug (or revision) in the near future is low. Hence, these variables model the scenario that files with recent bugs are more likely to have bugs in the future than others (see [Kim et al., 2007] and [Hassan and Holt, 2005]).

8.1.3 Choice of Algorithm and Performance Measurements

In this chapter we use two types of learning models: Class Probability Estimation (CPE) and regression models.

The CPE is a simple decision tree inducer (Weka's [Witten and Frank, 2005] J48 decision tree learner – a reimplementation of C4.5 [Quinlan, 1993]), which predicts the probability distribution of a given instance over the two possible classes of the target variable: `hasBug` and `hasNoBug`. Typically, one then chooses a cut-off threshold to determine the actual predicted class, which in turn can be used to derive a confusion matrix and the prediction's accuracy. We introduce misclassification cost when training a model such that both misclassification costs – false negative and false positive – are equal.

Component	#Files	Component	#Files
a11y	22	junit	106
accelerators	16	languages	124
ant	221	latex	322
antlr	58	lexer	198
apisupport	319	management	174
archivesupport	31	mdr	324
autoupdate	148	metrics	51
beans	52	mobility	1412
classclosure	6	monitor	99
classfile	55	nbbuild	111
clazz	26	nbi	278
cnd	1603	netbrowser	154
codecoverage	39	openide	941
collab	698	performance	343
contrib	1517	platform	107
corba	513	pluginportal	57
core	1000	portalpack	253
cpp	40	print	13
cpplite	100	projects	164
db	434	properties	41
debugercore	113	qa	184
debugerjpda	192	refactoring	212
debugertools	21	regsup	110
diff	72	remotefs	21
editor	843	rmi	72
enterprise	4261	ruby	314
extbrowser	46	schema2beans	94
externaleditor	20	scripting	204
form	472	serverplugins	1136
freestylebrowser	36	sim	204
graph	287	spellchecker	26
html	93	subversion	151
httpserver	16	tasklist	402
i18n	58	tomcatint	47
ide	183	treefs	47
innertesters	1	ui	65
installer	163	uml	3757
j2ee	2023	utilities	81
j2eeserver	126	vcsgeneric	239
jackpot	89	visualweb	2410
jasm	76	wasp	183
java	2085	web	614
javacvs	368	webl	11
javadoc	43	websvc	1107
jemmy	353	xml	2102
jemmysupport	23	xtest	209
jndi	73		

Table 8.3: Netbeans: Investigated components and number of files

Component	#Files	Component	#Files
accessibl	105	extension	328
browse	22	gf	101
buil	20	int	297
calenda	17	ip	38
camin	6	j	90
cap	6	mai	5
conten	388	mailnew	4
d	62	module	9
director	23	rd	5
do	28	suit	4
docshel	17	widge	5
edito	69	xpf	5
embeddin	242		

Table 8.4: Mozilla: Investigated components and number of files

Note that our datasets have a heavily skewed distribution (the ratio between defective files and non-defective ones is, depending on the project, about 1:20 and approximately remaining this ratio in all samples). For that reason we do not use the confusion matrix and associated accuracy as our performance measure as they are heavily influenced by this prior distribution. Instead we use the receiver operating characteristics (ROC) and the area under the ROC curve (AUC), which relate the true-positive rate to the false-positive rate and is independent of the prior distribution [Provost and Fawcett, 2001]. Note that an AUC close to 1.0 represents perfect and one close to 0.5 represents a random prediction quality.

For the regression experiments we use linear regression models. The linear regression is a form of regression analysis in which the relationship between one or more independent variables and another variable, called the dependent variable, is modeled by a linear function that minimizes the squared error of the weights associated with the independent variables. This function is a weighted linear combination of one or more model parameters, called regression coefficients. We report Pearson correlation, root mean squared error (RMSE), and mean absolute error (MAE) to measure the performance of the regression models.

We use the same WEKA APIs to implement the J48 decision tree and the linear regression models.

8.2 Method Implementation

In this section we explore the nature and possible causes for the variation in bug prediction quality. First, we show the variation in the defect prediction quality over time. In the second experiment we expand this finding and show that there exists periods of stability versus changes. To ensure that the observed phases of stability and variability in bug prediction quality are not random we next test their appearance statistically. Having ascertained that the explored projects

Name	Description
Features only from Versioning Control System (per file)	
activityRate	Number of revisions per month
lineAdded	# of lines added
lineDeleted	# of lines deleted
lineOperationRRevision	Number of line added and deleted per revision
revision	Number of revisions
totalLineOperations	Total # of lines added and deleted
chanceRevision	likelihood of a revision in the target period computed using $1/2^i$
Features only from Bugtracker	
blockerFixes	# of blocker type bugs fixed
blockerReported	# of blocker type bugs reported
criticalFixes	# of critical type bugs fixed
criticalReported	# of critical type bugs reported
enhancementFixes	# of enhancement requests fixed
enhancementReported	# of enhancement requests reported
majorFixes	# of major type bugs fixed
majorReported	# of major type bugs reported
minorFixes	# of minor type bugs fixed
minorReported	# of minor type bugs reported
normalFixes	# of normal type bugs fixed
normalReported	# of normal type bugs reported
trivialFixes	# of trivial type bugs fixed
trivialReported	# of trivial type bugs reported
p1-fixes	# of priority 1 bugs fixed
p1-reported	# of priority 1 bugs reported
p2-fixes	# of priority 2 bugs fixed
p2-reported	# of priority 2 bugs reported
p3-fixes	# of priority 3 bugs fixed
p3-reported	# of priority 3 bugs reported
p4-fixes	# of priority 4 bugs fixed
p4-reported	# of priority 4 bugs reported
p5-fixes	# of priority 5 bugs fixed
p5-reported	# of priority 5 bugs reported
Features from both CVS and Bugtracker	
chanceBug	Likelihood of a bug in the target period computed using $1/2^i$
lineAddedI	# of lines added to fix bugs
lineDeletedI	# of lines deleted to fix bugs
lineOperationIRbugFixes	Average number of lines operated to fix a bug
lineOperationIRTotalLines	# of lines operated to fix bugs relative to total line operated
lifeTimeBlocker	Average lifetime (avg. lt.) of blocker type bugs
lifeTimeCritical	avg. lt. of critical type bugs
lifeTimeMajor	avg. lt. of major type bugs
lifeTimeMinor	avg. lt. of minor type bugs
lifeTimeNormal	avg. lt. of normal type bugs
lifeTimeTrivial	avg. lt. of trivial type bugs
totalLineOperationsI	Total # of lines touched to fix bugs
grownPerMonth	Project grown per month (can be negative)
Target features	
hasBug (Target)	Indicates the location of a bug

Table 8.5: Extracted variables (features) from CVS and Bugzilla.

indeed exhibit statistically confirmed periods of stability and drift we investigate if we can predict the occurrence of such periods. Indeed a model predicting periods of stability and variability could be used to help understand the causes of those different phases. Consequently, we continue to explore two possible causes – author fluctuation and mean priority of bugs.

8.2.1 Does prediction quality varying over time?

The goal of this experiment is to show that the defect prediction quality varies over time. To that end, we conduct two experiments. In the first experiment, we keep the target period constant and predict defects on that target using the models trained on data collected from every possible combination of training periods. In the second experiment, we keep the prediction model constant and predict defects on varying target periods. As mentioned we use Weka's ([Witten and Frank, 2005]) J48 decision tree learner as a CPE induction method, which is trained with the features listed in Table 8.5. In both experiments the algorithm predicts the location of defects: *i.e.* it predicts which files will (or will not) contain bugs in the target period. The composition of datasets for these two experiments is as mentioned in Section 5.

For the first experiment, we start predicting defects on the last month – the target period – of the observed period of each project using the models trained from data collected on 2 months – the training period – before the target period. Next, we expand the training period by one month in order to collect more information still predicting on the same target period. This procedure is repeated until the training period reaches the maximum possible length into the past. Consequently, the maximum length for the training period in Eclipse and Netbeans is 74 months, for Mozilla 82 months, and for Open Office 85 months. Then, we move the target period one month backwards and repeat the above procedure. For example, if the initial target period is on the month t and the initial training period is $[t-1, t-2]$, then $[t-1, t-3]$, *etc.* Next we move the target period to $t-1$ and the initial training to $[t-2, t-3]$ and repeat the procedure.

For each training run we measure the model's prediction quality using its AUC value and visualize it using a heat-map. The resulting heat-maps are shown in Figures 8.1[2]–8.4[3]. In these heat-maps, the X-axis indicates the target period and the Y-axis the length of the training period in terms of number of months (*i.e.* for the training period $[t-n, t-m]$ Y is $m-n$).

We further compute the maximum, minimum, mean, and variance of the AUC values as well as the histogram of AUC variance's in each *column* of the heat-maps (Figures 8.1–8.4) and visualize them using bar charts as in Figures 8.5–8.8. In the bar charts 8.5a–8.5d, the x-axis shows the target period and the y-axis shows the AUC. In Figure 8.5e, the x-axis shows the bin values and the y-axis shows the frequencies. The x-axis and the y-axis of the bar charts of the Figure 8.5 are similar to the x-axises and the y-axises of corresponding bar charts of the Figures 8.6, 8.7 and 8.8.

[2]Note: In the first 9 months of Eclipse project there are no bug reports and therefore no prediction model was trained (blue area at the bottom left corner)

[3]Note: In the first 4 months there are no bug reports and therefore no prediction model was trained (blue area at the bottom left corner

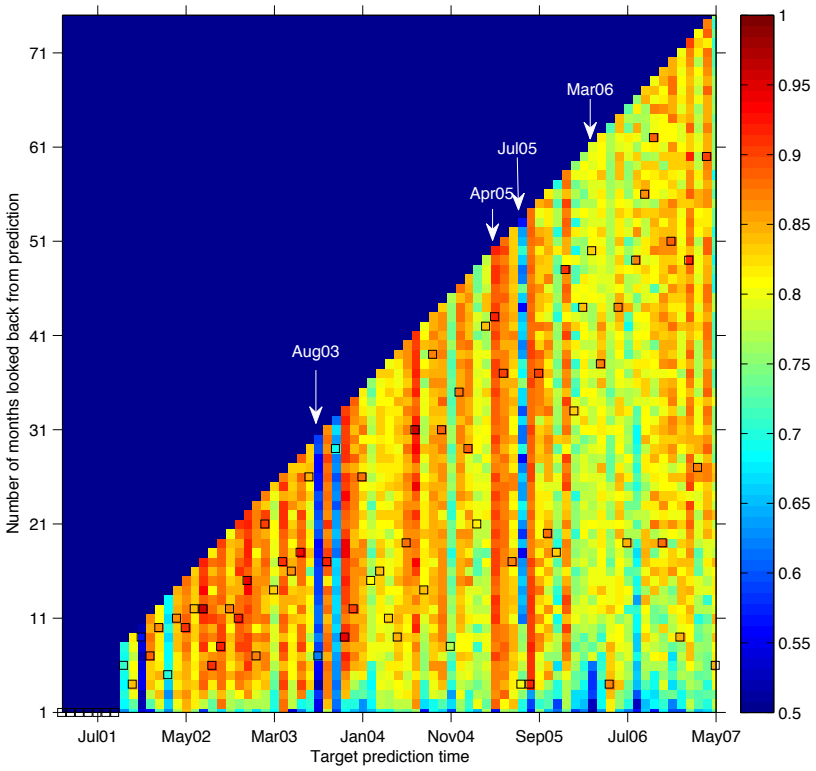


Figure 8.1: Eclipse heat-map: Prediction quality on same target using different training periods with the point of highest AUC highlighted

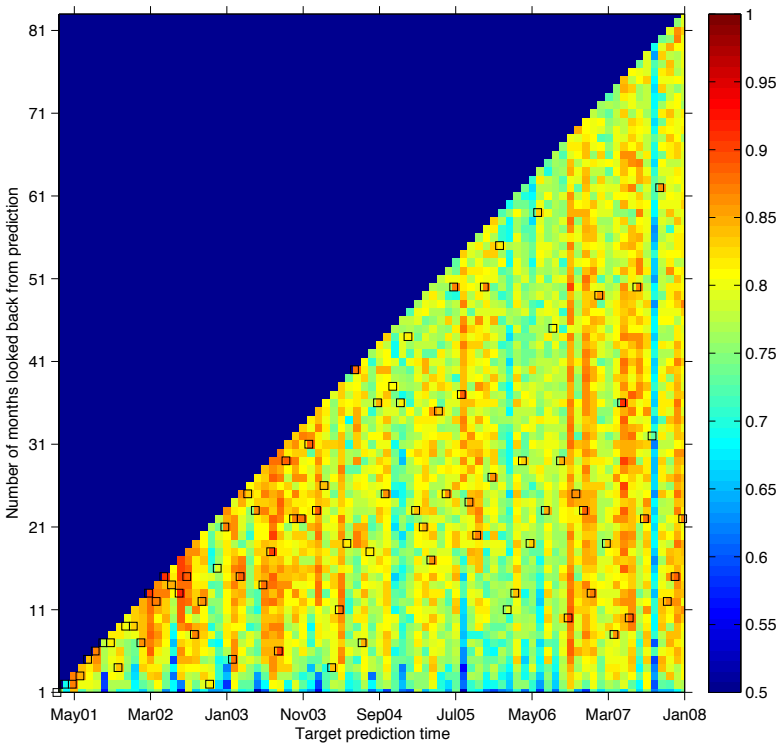


Figure 8.2: Mozilla heat-map: Prediction quality on same target using different training periods with the point of highest AUC highlighted

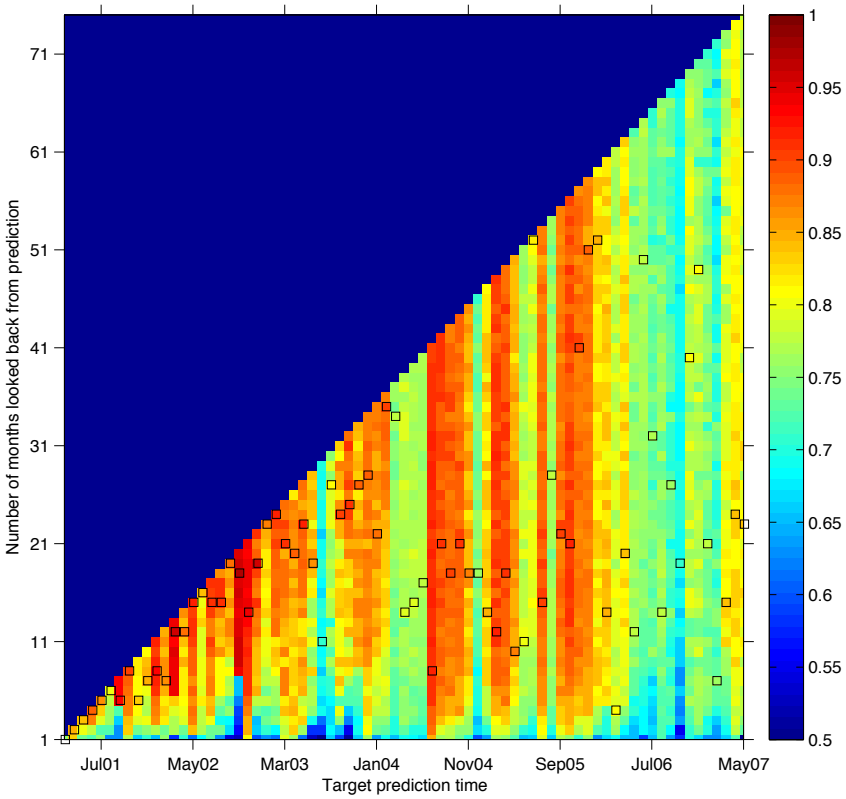


Figure 8.3: Netbeans heat-map: Prediction quality on same target using different training periods with the point of highest AUC highlighted

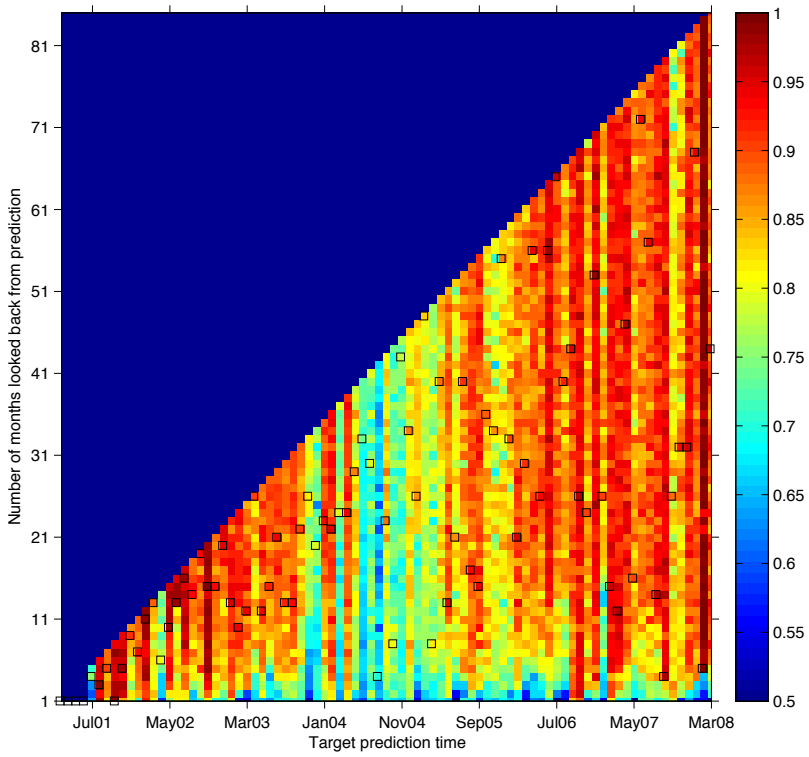
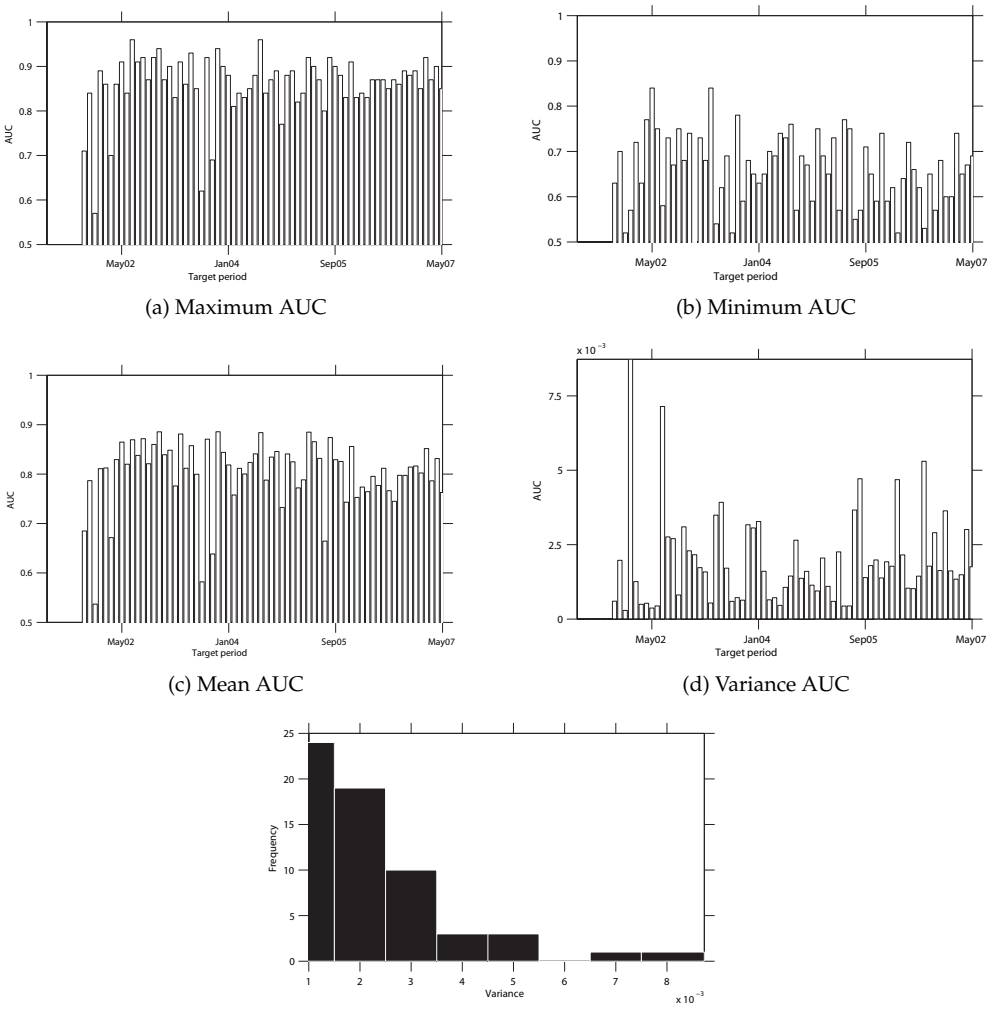


Figure 8.4: Open Office heat-map: Prediction quality on same target using different training periods with the point of highest AUC highlighted



(a) Maximum AUC

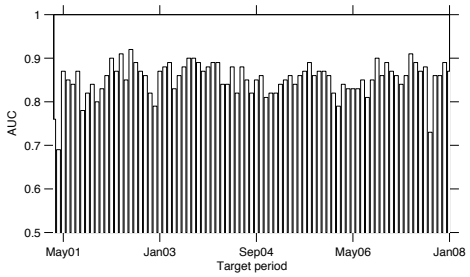
(b) Minimum AUC

(c) Mean AUC

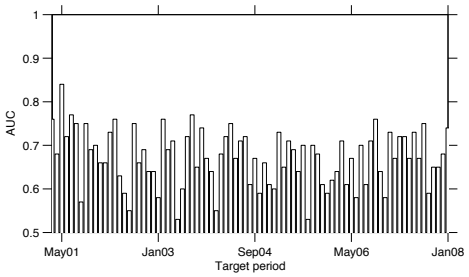
(d) Variance AUC

(e) Histogram: Variance

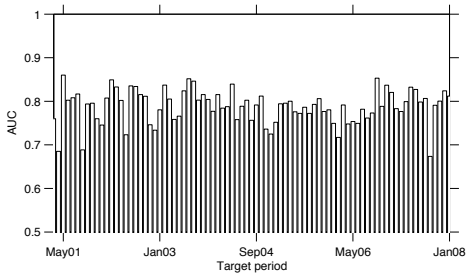
Figure 8.5: Descriptive statistics of AUC values in each column of the Eclipse heat-map (Figure 8.1)



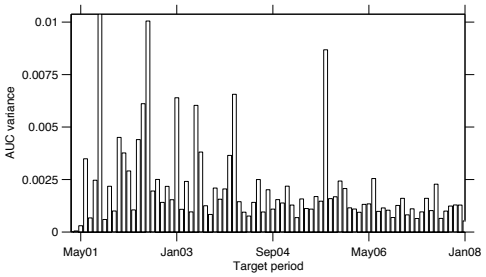
(a) Maximum AUC



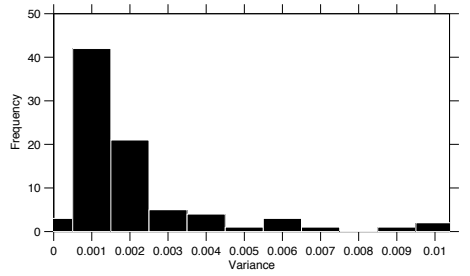
(b) Minimum AUC



(c) Mean AUC

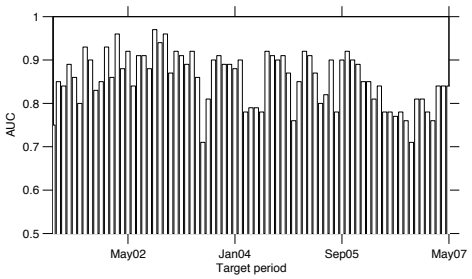


(d) Variance AUC

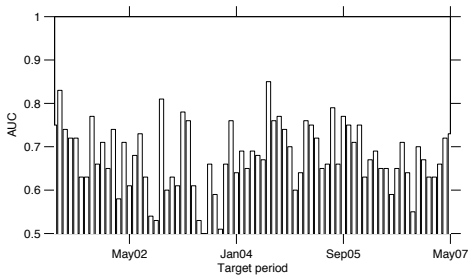


(e) Histogram: Variance

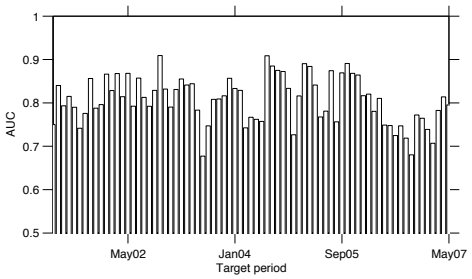
Figure 8.6: Descriptive statistics of AUC values in each column of the Mozilla heat-map (Figure 8.2)



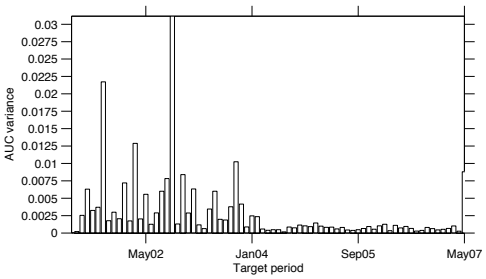
(a) Maximum AUC



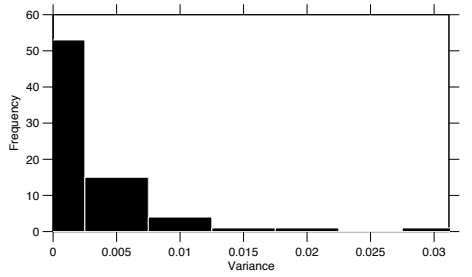
(b) Minimum AUC



(c) Mean AUC

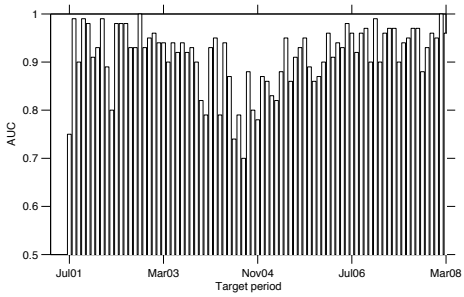


(d) Variance AUC

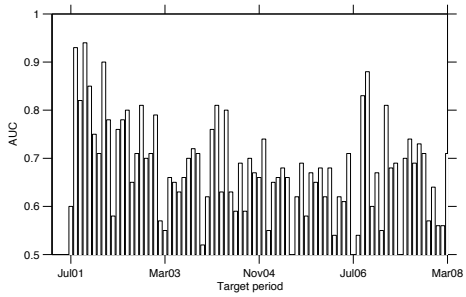


(e) Histogram: Variance

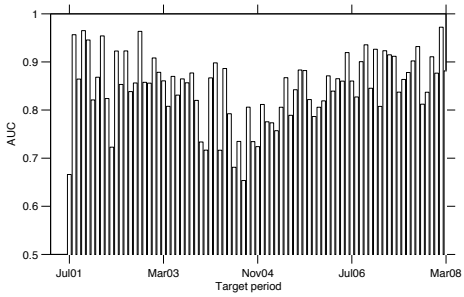
Figure 8.7: Descriptive statistics of AUC values in each column of the Netbeans heat-map (Figure 8.3)



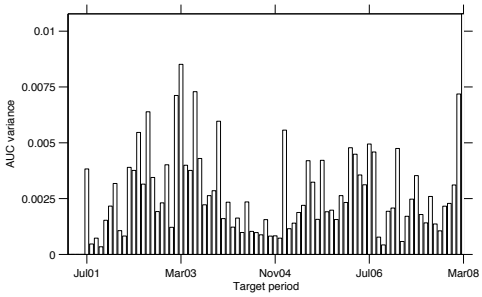
(a) Maximum AUC



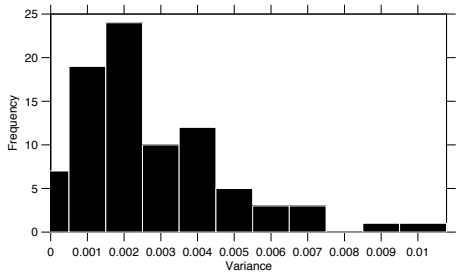
(b) Minimum AUC



(c) Mean AUC



(d) Variance AUC



(e) Histogram: Variance

Figure 8.8: Descriptive statistics of AUC values in each column of the Open Office heat-map (Figure 8.4)

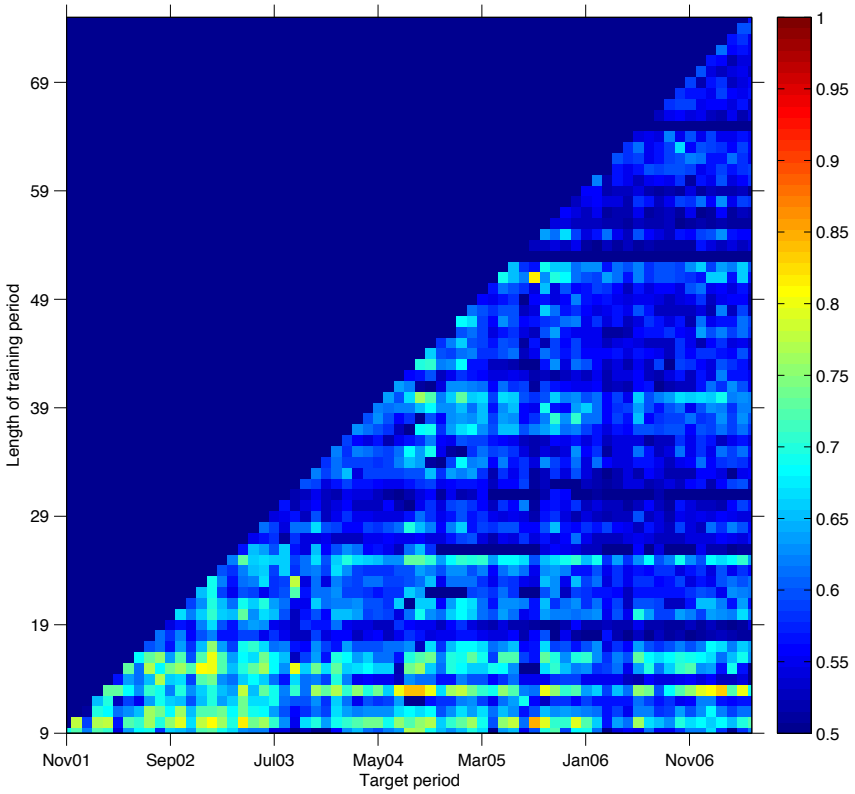


Figure 8.9: Eclipse heat-map: Prediction quality at different target periods

What we have done so far is that we keep the target constant but vary the training period. In the second experiment we establish that the prediction quality varies when we keep the training period constant and change the target. To that end our second experiment initially trains a prediction model using the data collected from the first two months of the observed period and then uses this model to predict defects on the third month, fourth month, until the last month of the observed period. Next we expand the training period by one month and start predicting defects from the fourth month onward. This procedure is repeated until the training periods reach the maximum observation period. Similar to the above experiment, we measure model's prediction quality for each target period using AUC value and visualize all of these values in a heat-map (see Figures 8.9–8.12); x- and y-axis are the same as in Figures 8.1–8.4.

Analogous we compute the descriptive statistics of the AUC values in each *row* of these heat-maps and visualized in bar charts as in Figures 8.13–8.16. In the bar charts 8.13a–8.13d, the x-axis shows the length of the training period and the y-axis shows the AUC values. The x and y-axises of bar chart 8.13e is same as in the above experiment. The x and y axises of other Figures (8.14,8.15

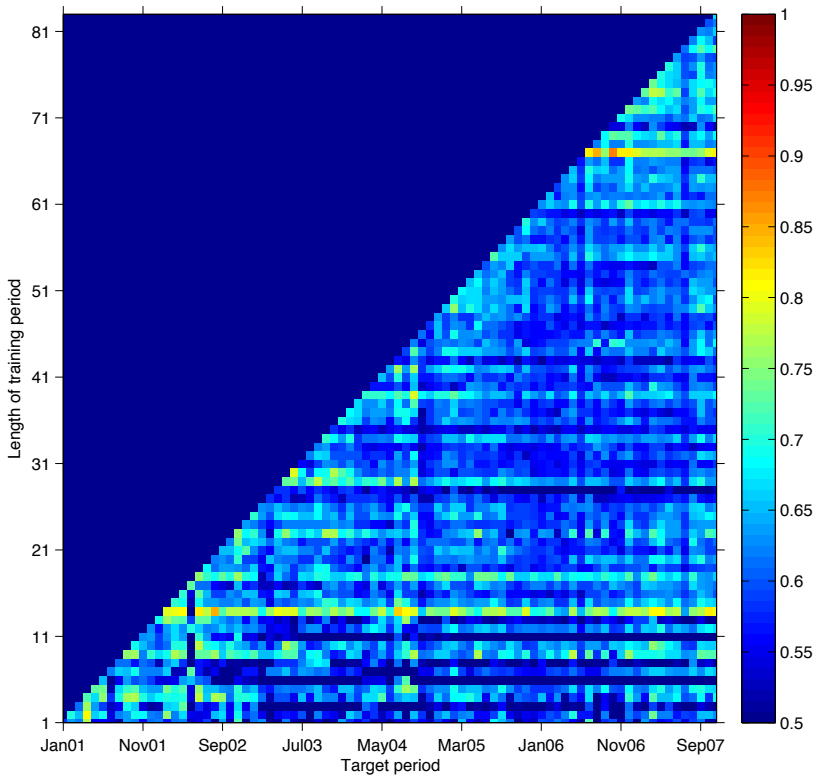


Figure 8.10: Mozilla heat-map: Prediction quality at different target periods

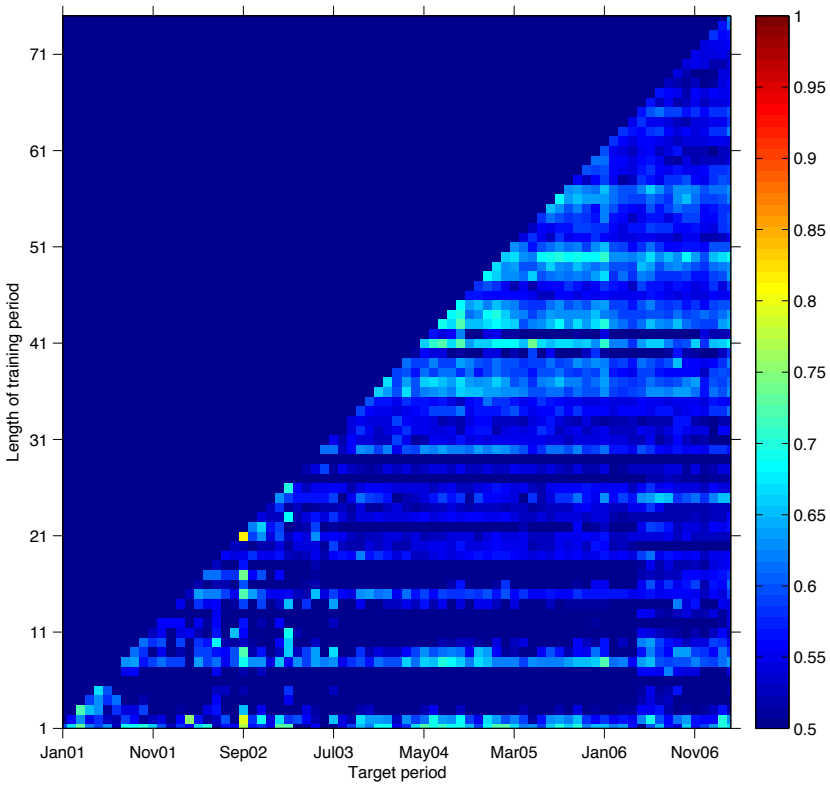


Figure 8.11: Netbeans heat-map: Prediction quality at different target periods

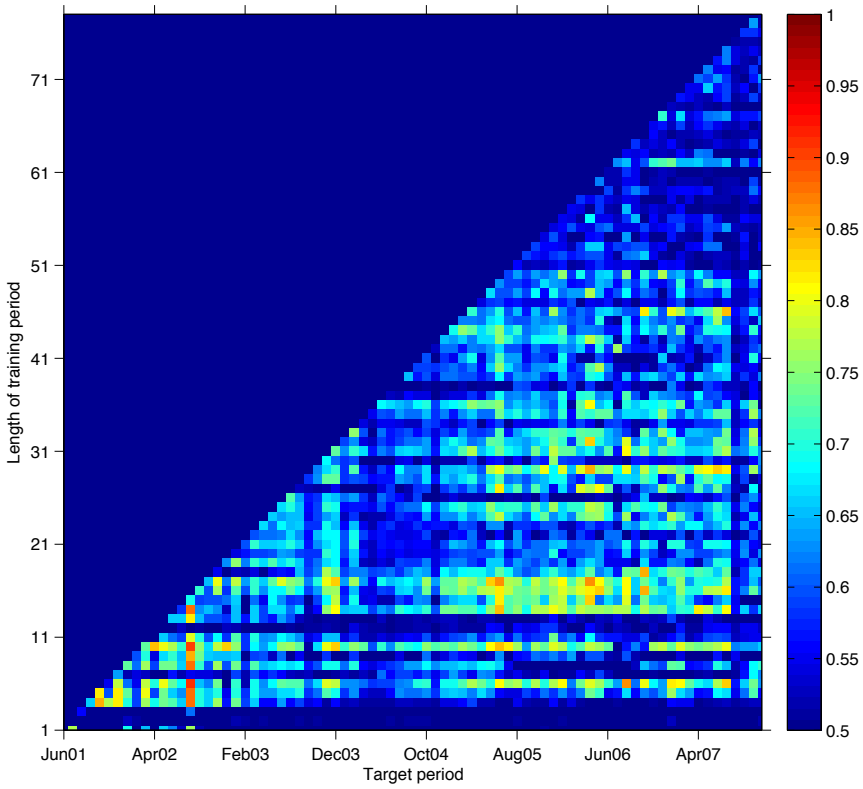


Figure 8.12: Open Office heat-map: Prediction quality at different target periods

and 8.16) have the same meaning as Figure 8.13.
Following paragraph discusses the outcome of theses two experiments.

Results and Discussion

According to the observations in Figure 8.1 generated from Eclipse data, the models involved in predicting defects on certain target periods (*e.g.,* April 2005) obtain an AUC around 0.9 (see Mean AUC in Figure 8.5c) while the models that predict defects in August 2003 obtain an AUC around 0.6. In some prediction periods (*e.g.,* March 2006) the prediction quality is initially relatively low but when expanding the learning period up to certain months back, the models gain prediction quality. Contrastingly in other cases (*e.g.,* July 2005), a further expansion of the training period causes a degradation of prediction quality. The maximum AUC values for each target period (shown as square in Figure 8.1–8.4) typically lie on neither ends. This suggests that in order to obtain higher prediction accuracy, the models should not be trained on data collected from very long or very short history. We can find similar observations in other three projects as well.

Hence, models at different time periods (and varying length) seem to vary, but do they do so significantly? To establish that the AUC varies significantly we compare the distribution of high and low AUC-variance values. Specifically, we use the split-half method as described by [Ko and Chilana, 2010]: First rank the variance values in descending order and then divide it into two equal parts. Having tested the normality of the data using one-Sample Kolmogorov-Smirnov (K-S) test we conducted a parametric or non-parametric test to compare high and low half of the AUC-variance values and the results are listed in Tables 8.6–8.9.

So far we have seen that the prediction quality is varied due to changes in training periods. The same variation can be observed due to change in target periods (see Figures 8.9–8.12). Again, the split-half method on the variance values of AUC shows a significant difference between the high and low AUC-variance values (see Tables 8.10–8.13 for details).

Note, however, that this experiment does not address the question of establishing the optimal training period – a question we leave open for future work. It is also important to note that the models trained in different training periods are likely to rely on different predictors.

Summarizing, these two experiments show that the prediction quality varies over time: both when holding the model constant and predicting varying target periods (change along the x-axis in Figures 8.9–8.12) as well as when sliding the training period while predicting the same target (change along the y-axis in Figures 8.1–8.4). Hence, models that are good predictors in some target periods are likely to be bad ones on others and the prediction quality of models on a given target period vary based on the training period.

8.2.2 Finding Periods of Stability and Change

So far we have seen that the prediction quality varies over time. But, are there periods that the prediction quality is good and continue this trend for a period of stability or periods of change

	#of values	Mean	P-value
Upper half	34	0.003	0.000
lower half	34	0.0009	

Table 8.6: t-test on high and low AUC variance values (column) of Eclipse heat-map (Figure 8.1): p-value of K-S test for normality is 0.055

	#of values	Mean Rank	P-value
Upper half	42	63.0	0.000
lower half	41	21.5	

Table 8.7: Mann-Whitney test on high and low AUC variance values (column) of Mozilla heat-map (Figure 8.2): p-value of K-S test for normality is 0.000

	#of values	Mean Rank	P-value
Upper half	42	55.0	0.000
lower half	41	17.5	

Table 8.8: Mann-Whitney test on high and low AUC variance values (column) of Netbeans heat-map (Figure 8.3): p-value of K-S test for normality is 0.000

	#of values	Mean	P-value
Upper half	42	0.0042	0.000
lower half	41	0.0012	

Table 8.9: t-test on high and low AUC variance values (column) of Open Office heat-map (Figure 8.4): $p - value$ of K-S test for normality is 0.065

	#of values	Mean	P-value
Upper half	33	0.077	0.000
lower half	34	0.035	

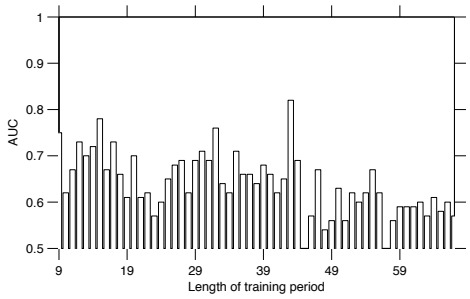
Table 8.10: t-test on high and low AUC variance values (row) of Eclipse heat-map (Figure 8.9): $p - value$ of K-S test for normality is 0.442

	#of values	Mean	P-value
Upper half	41	0.086	0.000
lower half	42	0.042	

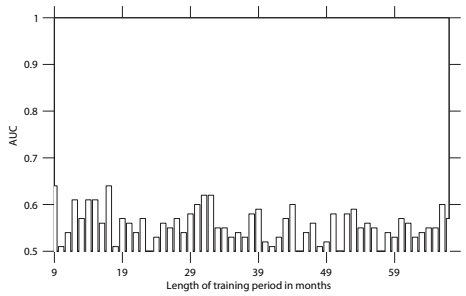
Table 8.11: t-test on high and low AUC variance values (row) of Mozilla heat-map (Figure 8.10): $p - value$ of K-S test for normality is 0.088

	#of values	Mean	P-value
Upper half	37	0.0775	0.000
lower half	38	0.0300	

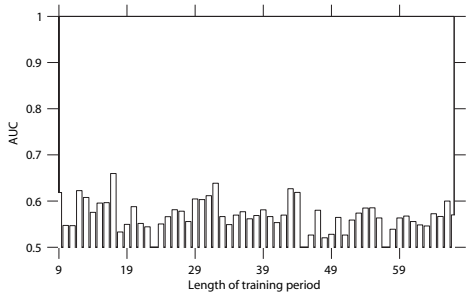
Table 8.12: t-test on high and low AUC variance values (row) of Netbeans heat-map (Figure 8.11): $p - value$ of K-S test for normality is 0.789



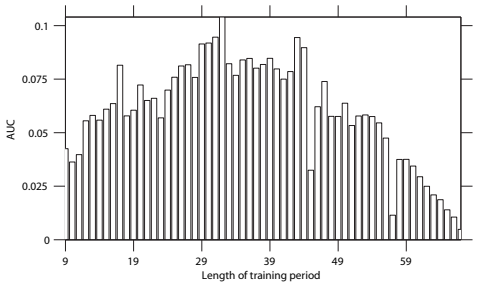
(a) Maximum AUC



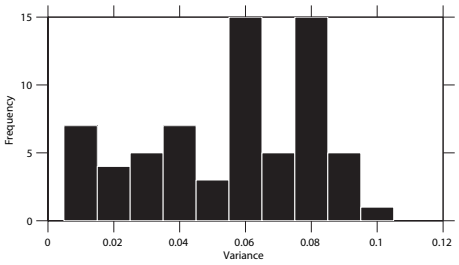
(b) Minimum AUC



(c) Mean AUC



(d) Variance AUC

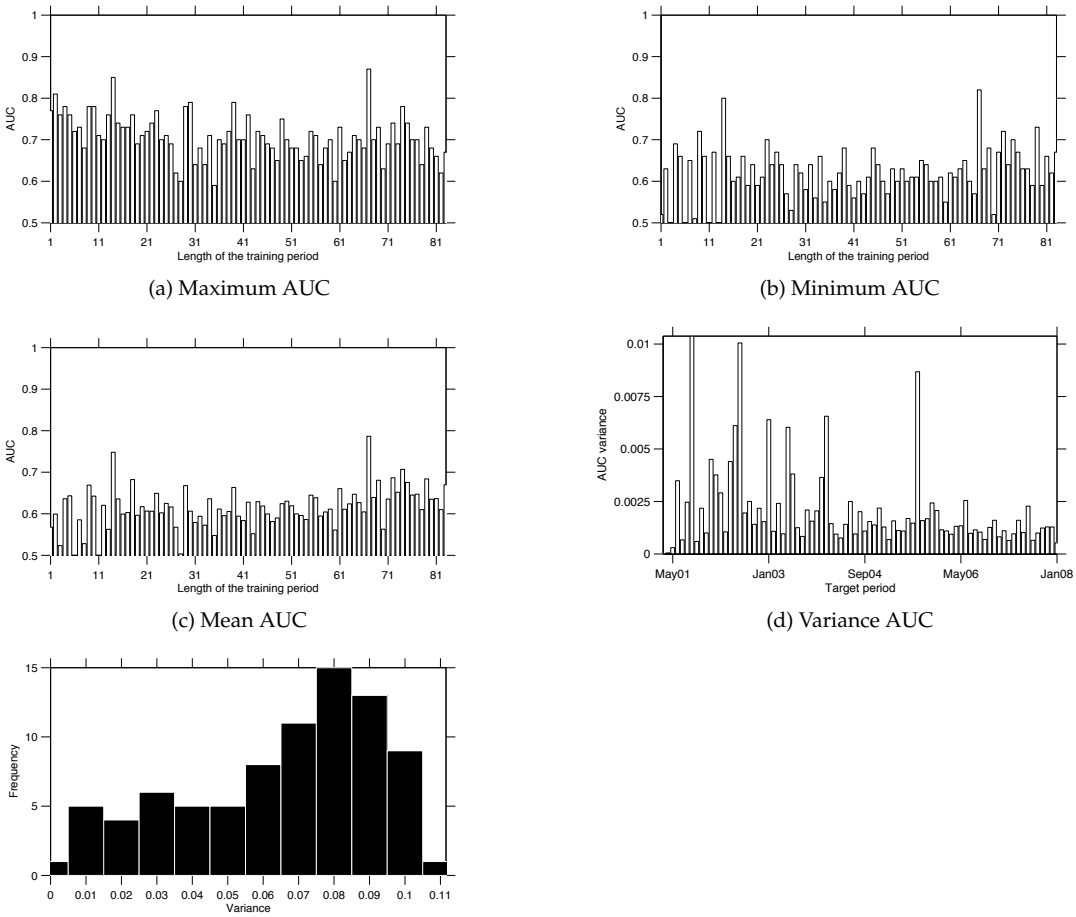


(e) Histogram: Variance

Figure 8.13: Descriptive statistics of AUC values in each row of the Eclipse heat-map (Figure 8.9)

	#of values	Mean	P-value
Upper half	38	0.0884	0.000
lower half	39	0.0377	

Table 8.13: t-test on high and low AUC variance values (row) of Open Office heat-map (Figure 8.12): $p - value$ of K-S test for normality is 0.599



(a) Maximum AUC

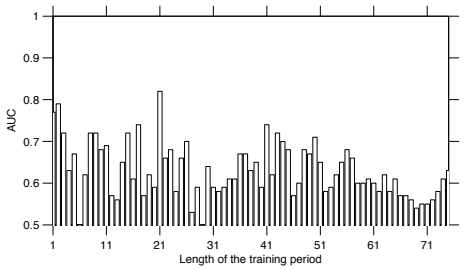
(b) Minimum AUC

(c) Mean AUC

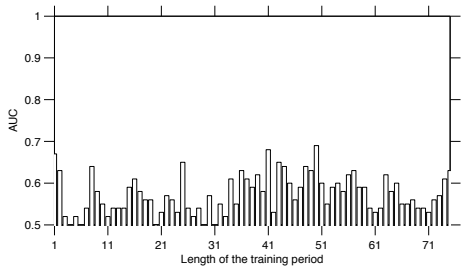
(d) Variance AUC

(e) Histogram: Variance

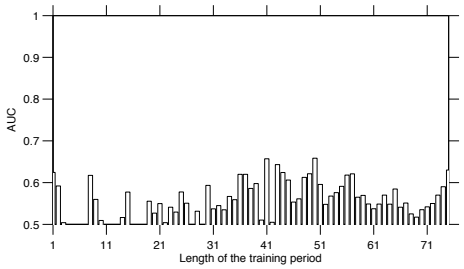
Figure 8.14: Descriptive statistics of AUC values in each row of the Mozilla heat-map (Figure 8.10)



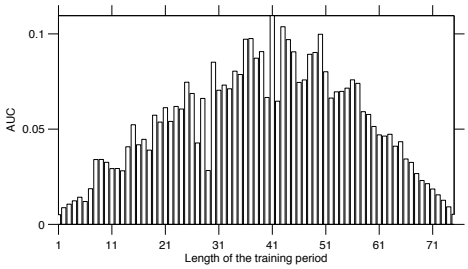
(a) Maximum AUC



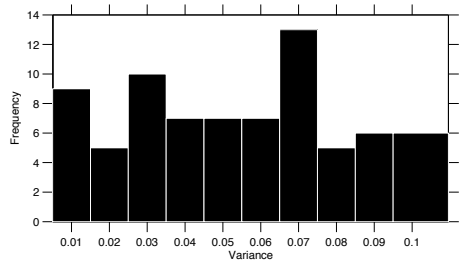
(b) Minimum AUC



(c) Mean AUC

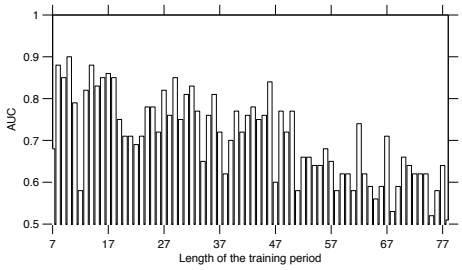


(d) Variance AUC

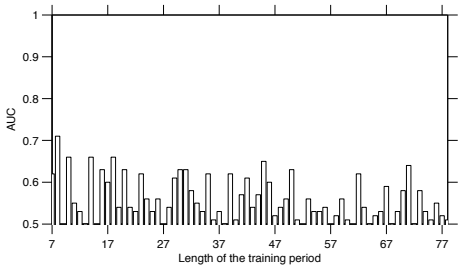


(e) Histogram: Variance

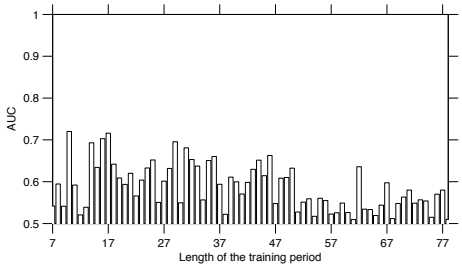
Figure 8.15: Descriptive statistics of AUC values in each row of the Netbeans heat-map (Figure 8.11)



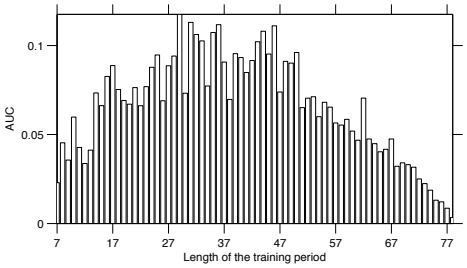
(a) Maximum AUC



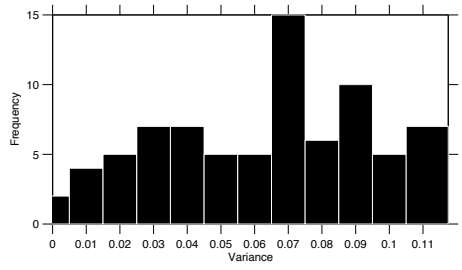
(b) Minimum AUC



(c) Mean AUC



(d) Variance AUC



(e) Histogram: Variance

Figure 8.16: Descriptive statistics of AUC values in each row of the Open Office heat-map (Figure 8.12)

where the model's prediction quality changes continuously?

To differentiate periods of stability and change we slightly adapted our experiment as follows: similar to the first of the experiments above, we kept the target period constant but varied the training period. In contrast to that experiment, we used a two-month training window and slided this training window into the past. The composition of datasets for this experiment is as mentioned in Section 5, but the only difference is that the length of the training period is two months. Following is the formal definition of the dataset.

Assume that the observed period is d months. $Y_T = \{y_{T,1}, y_{T,2}, ..., y_{T,j}, ..., y_{T,s}\}$ is a vector of dimension s (s is the number of observed files) and $y_{T,j}$ is the number of bugs reported for file j at time T, where $2 < T \leq d$ and T is one-month labeling period. if $X_t = \{f_{t,1}, f_{t,2}, , f_{t,i}, ..., f_{t,n}\}$ is a feature vector of dimension n and $f_{t,i}$ is a file feature i computed from two-months training window that is t months backwards from the labellings period, where $n \in N$ and $1 < t \leq d-1$, $t < T$ and $s >>> n$, then constructed dataset is given by X_t, Y_T. By changing the t and T variables we can generate different datasets. This dataset is used to train models for predicting defects on time $T+1$.

We used a two-months training window because the typical release cycle of the considered projects is 6 to 8 weeks. In addition, Bernstein *et al.* [Bernstein et al., 2007] have shown that 2 months of history data attains higher prediction quality. we employed Weka's J48 decision tree learner.

Figures 8.17[4],–8.20[5] visualize the results of this procedure for the considered projects. Note that whilst the X-axis of these graphs shows the target period as before, the Y-axis has a different meaning: it represents the time-difference between the target period and the two-month training window in months. Hence, the higher in the figure we are looking at, the older the two-month period is compared to the target. Values on the diagonal (bottom left to top right) from each other represent predictions of the model trained on the same period.

Results and Discussion

By looking at the the heat-maps (Figures 8.17–8.20) we can see some triangle shapes (**green color**). For instance, in Figure 8.17, one such triangle starts from April 2002 and continues till July 2003. During these periods the prediction quality stayed relatively stable and a triangle seemingly emerges as the old training data (along the upper left boundary/diagonal of the triangle) remains predictive. But what is a "good" prediction quality?

To identify periods of stably good predictions and maintain that the triangles indicate periods of stably "good" predictions we need a notion of what "good" predictions are. Whilst the AUC scale clearly has some boundaries for "perfect" (=1) and "random" (=0.5) it is not necessarily clear

[4]Note: In the first 9 months there are no bug reports in the target period and therefore no prediction model was built (white area at the bottom left corner)

[5]Note: In the first 4 months there are no bug reports in the target period and therefore no prediction model was built (white area at the bottom left corner)

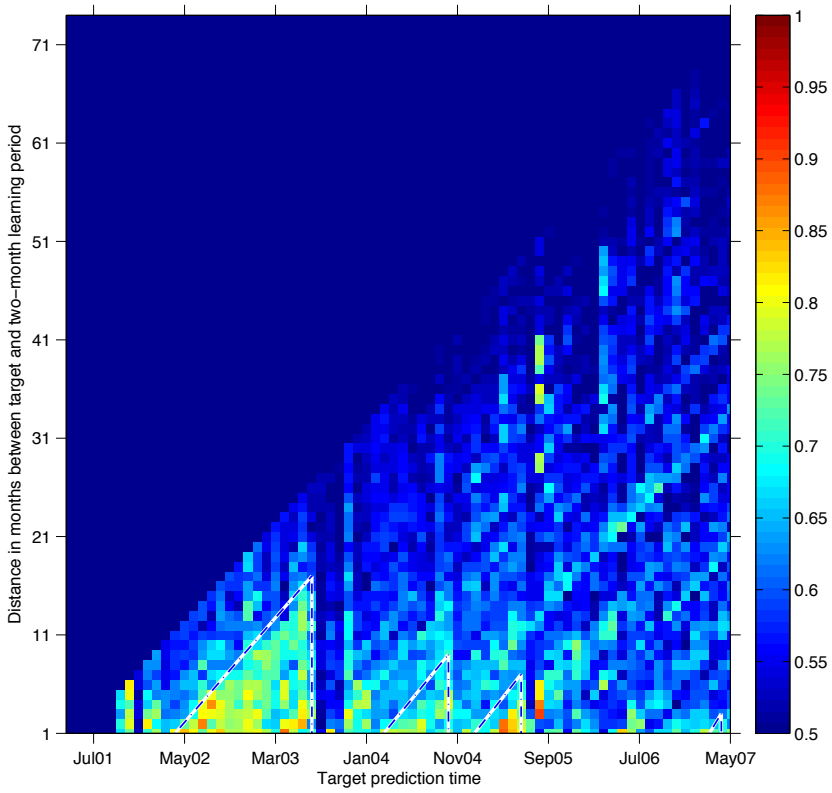


Figure 8.17: Two-month Heat-map: Eclipse

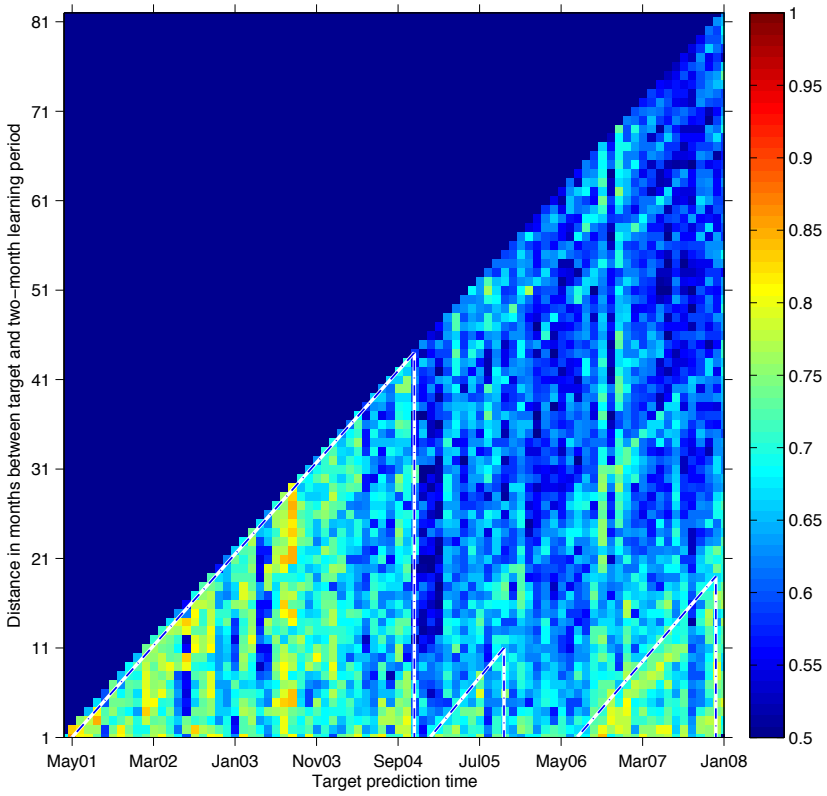


Figure 8.18: Two-month Heat-map: Mozilla

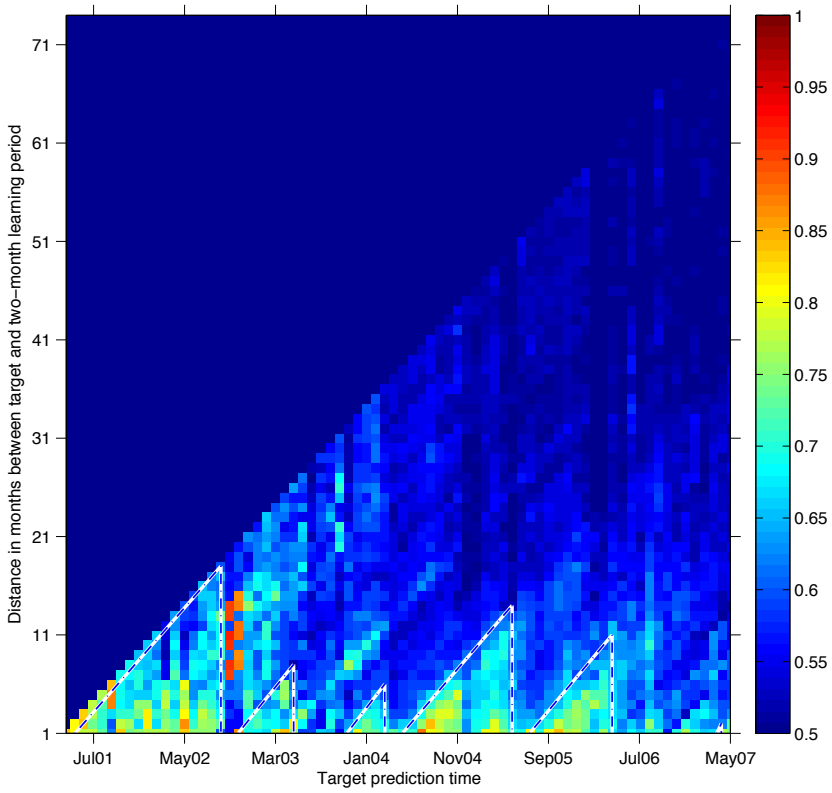


Figure 8.19: Two-month Heat-map: Netbeans

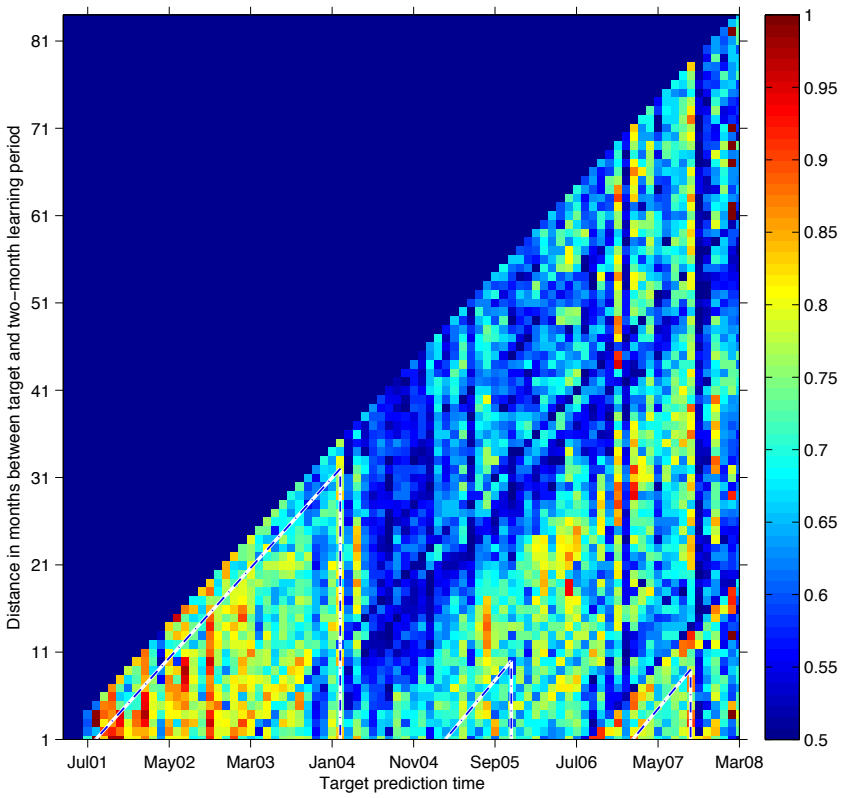


Figure 8.20: Two-month Heat-map: Open Office

Cache size%	min	max	median	mean
10	0.45	0.66	0.51	0.52
15	0.40	0.69	0.51	0.52
20	0.40	0.74	0.52	0.54
25	0.40	0.76	0.55	0.57

Table 8.14: Prediction quality (AUC) of the model [Kim et al., 2007] for Eclipse project; Observed period is Apr 2001-Jan 2005

what can be regarded as "decent" or "sufficiently good" in any particular task?

To determine a notion of "decent" for our task empirically we first determined the attainable prediction quality on our data using the award winning BugCache prediction model [Kim et al., 2007]. We ran BugCache on our Eclipse project data (the observed period is from April 2001 to January 2005) with different cache sizes and present the results in Table 8.14. The table shows the minimum, maximum, median, and mean of AUC for each of the runs. As the table shows the maximum attained AUC varies between 0.66 for the smallest cache and 0.76 for the largest one. Given BugCache's usual prediction quality we decided to take the lower end of these values as indicating "sufficiently good" and set our threshold for "decent" predictions on our data to AUC=0.65.

To illustrate the resulting triangle shapes Figures 8.17–8.20 indicate periods where more than 80% of the values are higher than the threshold with a drawn triangle. Consequently, we find that the prediction periods inside the triangles are stable even on models learned from older data. Returning to the stable example period in Eclipse (April 2002 - July 2003 in Figure 8.17) we find that even data from the second quarter of 2002 (more that a year old) provides a decent prediction quality.

In all figures we also observe that the further we move the training period to the past, the more likely the prediction quality would drop down to almost random (≈ 0.5). This provides some evidence to the statement *the further back you go in time the more the prediction deteriorates* ([Kenmei et al., 2008]). More formally, from April 2002 to July 2003 the model exhibits a stable good prediction quality. In March 2004 the project seems to recover some stability in defect prediction quality and generate another, but slightly less pronounced triangle until October 2004. The triangles exhibited by the Netbeans project look similar to the one of Eclipse: relatively small (approximately 1 year) but with a high frequency. Mozilla and OpenOffice, on the other hand, have long periods of stability (e.g., Mozilla: from May 2001 until November 2004). In such a period, a two-month training window, which is older than 3 years can predict defects with decent accuracy of AUC around 0.7.

Summarizing, the model exhibits periods of stability and variability in defect prediction quality over time. The causes of the changes – be they observable in our features or not – are not obvious from the graphs and will be investigated in Section 8.2.4. Another interesting observation in the heat-maps is the height of the triangle-shapes. It indicates the length of the stable period. Note that the height varies both within and between projects. Hence, a universal optimal

training period length can not be determined but it is highly dependent on the causes for the current stable period. Finally, this finding indicates that decision makers in software projects should be cautious to base their decisions on a generic defect prediction model. Whilst they might be useful in periods of stability they should be ignored in periods of variability.

All of these findings assume that the triangle shapes observed are indeed a feature of the underlying software projects rather than a random artifact – a question to which we turn in the next subsection.

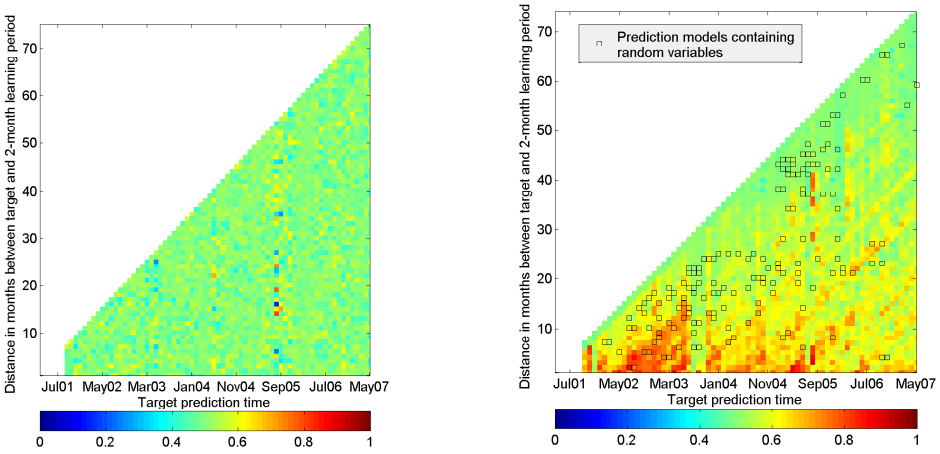
8.2.3 Triangle Shapes are not Random Phenomena

To illustrate that the triangle shapes are not an epiphenomenon of the data or the prediction algorithm, we graphed the result of a naïve model. In the naïve model we simply assume that if a file in the target period recorded at least one bug in the given two-month training period, then we predict that that file is going to be buggy in the target period. As Figure 8.21a clearly shows for Eclipse, most predictions attained in this manner are random (*i.e.* AUC ≈ 0.5; white in the figure) and do not exhibit the triangle shapes.

To elicit if that the triangles indeed visualize a phenomenon of the underlying data rather than the prediction process itself, we added 10 random variables to our feature set. The random features are generated from similar distributions as 10 real variables that are selected randomly. To avoid an outlying result we repeated this procedure 4 times. In each of these runs, the number of models that actually picked up the random features was between 158 to 162 of the total 2850 models computed.[6] Figure 8.21b shows the run with 162 models picking random features; the other runs look almost identical. It is important to note that the models containing random features (marked with a square in the figure) are mostly found when the AUC is close to 0.5 (*i.e.* random). In only 14% of the cases does a model with AUC > 0.65 pick up a random variable. Hence, we can assume that they are mostly picked due to the noise in the data. Models pick a maximum of 2 random features, which appear lower than the 3rd level in the decision tree. Hence, the random features seem to be seldomly used by predictive models (where AUC $\gg 0.5$) do not seem to be dominating in those models. But do they deteriorate the predictive quality of those models?

To show that the random features have no statistically significant effect on the prediction quality, we compare the prediction quality of the models with the random features (see Figure 8.21b) to the ones without (see Figure 8.17). To that end, we first determine triangles in Figure 8.21b (with random feature) using the same method described in the previous section. Further, we understand that these triangles are located in the same places as in Figure8.17. We then generate pairs of AUC values as follows: We pick one AUC value from the triangles in Figure 8.17 and the other AUC value from Figure 8.21b at the same coordinate. Having tested the normality of the data with Shapiro-Wilk test ($\alpha = 0.05, p = 0.0174$) we performed a pairwise t-test comparing those selected values and found them to be significantly similar ($p = 0.0046$).

[6]Note that the observed number of models (162) that pick random features significantly differ from the expected number of models (1425) according to χ^2-test at ($p < 0.001$).



(a) Naïve prediction model

(b) Usage of random features

Figure 8.21: Experiments to exclude the possibility of the triangles being an epiphenomenon of the data or the prediction algorithm. (Eclipse data)

The above experiments show that our feature set is better than a set of random features and the quality of prediction models (*i.e.* the ones within the triangles) is not significantly different in the presence or absence of random features. Hence, the triangles must be a result of the underlying models' predictive power given the available data. Having found that our observation of periods of stability and variability is sound, we turn to try to identify the causes for such variability and set the stage for making these observations actionable.

8.2.4 Finding Indicators for Prediction Quality Variability

In Section 8.2.2 we show that defect prediction models exhibit periods of stability and change. Can we uncover reasons for such variability? To that end we learned a regression model to predict the AUC of the bug prediction model according to the following procedure:

First, we computed the AUC of the bug prediction model based on the data in the three months (two months training period and one month labeling period as described in Section 8.2.2) before the target period in exactly the same way as in the previous subsection. The AUC is derived using the file-level features but is a feature of predicting defects within a project. Hence, it is a project-level feature.

Secondly, since the AUC is a project-level feature, we needed project-level features to train a prediction model. Thus, we computed a series of project-level features (listed in Table 8.15) by aggregating the respective file-level features in two-months training windows and 1-month labeling periods. However, we do not include the information about the labeling period in the features since it is unique and consequently not a useful predictor. We also exclude the distance

between the target period and the two-months training window since the finding *the further you go back in time worst will be the prediction* is not novel for the software engineering community.

Thirdly, since (i) the AUC prediction model used a two-month training period and (ii) we are interested in changes between the training and the labeling period we transformed the features by taking the average of the two training months ($avg_t = average(feature_{t-1}, feature_{t-2})$) and subtracting it from the values of the labeling month ($= feature_t - avg_t$).

Fourth and last, we trained a traditional linear regression model predicting the AUC from these transformed features.

Results and Discussion

The resulting regression models are shown in Tables 8.16, 8.17, 8.18, and 8.19. If a regression coefficient is large compared to its standard error, then it is probably different from zero. The p-value of each coefficient indicates whether the coefficient is significantly different from zero, such that if it is ≤ 0.05 (with 95% confidence interval) then those variables significantly contribute to the model, else there is no significant contribution of those variables. We did not list those coefficients that are not significant. The performance of the models as measured in terms of their Pearson correlation, Spearman's rank correlation, mean absolute error (MAE), and root mean square error (RMSE) is shown in Table 8.20. Note that all models have a moderate correlation between the predicted and actual values of AUC. The small MAE and RMSE reflect the good performance of our regression models.

In all regression models the change in the *number of authors* feature (for brevity we call this `author` in the tables) has a negative impact on the AUC. Thus if the number of authors in the target period is larger than the number of authors in the learning period then the defect prediction quality goes down and vice versa. Hence, addition of more authors to a project reduces the applicability of the defect prediction model learned without those authors. One possible reason could be that adding more authors to a project may lead to a change in the underlying development patterns in the software project. In order to comply with the new state of the project, a model should be learned with the new information of the project.

The regression coefficients (unstandardized) for `author` in all four models are very small, but since the AUC moves in the range of $0 - 0.9$ they contribute about 1% to the model providing at least a qualitative indication. For example in Eclipse: including 10 more authors in the target period than in the leaning period will decrease the AUC by 0.065 and this is a considerable amount of decrease in AUC. For the other projects, the influence of author attribute is an order of magnitude smaller but still pointing in the right direction. For OpenOffice, the effect is statistically insignificant at the 98% confidence interval. Hence, the effect is observable in most projects but is most pronounced in the Eclipse.

Another interesting feature of the models is the number of lines added/removed to fix bugs relative to the total number of lines changed (called `LineOpeIRTotLines`). This feature reflects the fraction of work performed to fix bugs relative to total work done. In all models except

Name	Description
revision	Number of revisions
grownPerMonth	Project grown per month
totalLineOperations	# of lines added and deleted
bugFixes	# of bugs fixed (all types)
bugReported	# of bugs reported (all types)
enhancementFixes	# of enhancement requests fixed
enhancementReported	# of enhancement requests reported
p1-fixes	# of priority 1 bugs fixed
p2-fixes	# of priority 2 bugs fixed
p3-fixes	# of priority 3 bugs fixed
p4-fixes	# of priority 4 bugs fixed
p5-fixes	# of priority 5 bugs fixed
p1-reported	# of priority 1 bugs reported
p2-reported	# of priority 2 bugs reported
p3-reported	# of priority 3 bugs reported
p4-reported	# of priority 4 bugs reported
p5-reported	# of priority 5 bugs reported
lineAddedI	# of lines added to fix bugs
lineDeletedI	# of lines deleted to fix bugs
totalLineOperationsI	# lines operated to fix bugs
lineOperIRbugFixes	Average (avg.) # of lines operated to fix a bug
lineOperIRTotalLines	# of lines operated to fix bugs relative to total line operated
lifeTimeIssues	avg. lifetime of bugs (all types)
lifeTimeEnhancements	avg. lifetime of enhancement type bugs
authors	# of authors
workload	avg. work done by an author
AUC(Target)	Area under ROC curve

Table 8.15: Project level features for regression

Feature	**Unstand:**	**Standard:**	p
(Constant)	0.67		0.000
enhancementFixes	0.0002	0.168	0.000
enhancementReported	0.0001	0.125	0.004
p1-fixes	-0.0013	-0.494	0.000
p3-fixes	-0.0002	-0.481	0.000
p5-fixes	-0.043	-0.071	0.001
p1-reported	0.0015	0.54	0.000
p2-reported	0.0001	0.118	0.000
p3-reported	-0.0001	-0.09	0.023
p4-reported	-0.0005	-0.069	0.000
p5-reported	-0.005	-0.145	0.000
LineOperIRbugFixes	-0.001	-0.264	0.000
LineOperIRTolLines	-0.1127	-0.244	0.000
author	-0.0065	-0.324	0.000

Table 8.16: Eclipse: Regression Model

Feature	**Unstand:**	**Standard:**	p
(Constant)	0.7333		0.000
revision	-0.0001	-0.6	0.000
bugFixes	0.0001	0.722	0.000
enhancementFixes	-0.0012	-0.264	0.000
enhancementReported	-0.0004	-0.098	0.000
p1-fixes	0.0004	0.153	0.000
p2-fixes	0.0003	0.221	0.000
p3-fixes	0.0003	0.15	0.000
p4-fixes	0.0012	0.185	0.000
p5-fixes	-0.0016	-0.393	0.000
p3-reported	0.0007	0.202	0.000
p4-reported	0.0005	-0.073	0.001
p5-reported	0.001	0.087	0.000
LineOperIRbugFixes	0.0011	0.396	0.000
LineOperIRTolLines	-0.2478	-0.220	0.000
author	-0.0007	-0.137	0.001

Table 8.17: Mozilla: Regression Model

Feature	**Unstand:**	**Standard:**	p
(Constant)	0.67	0.000	
bugFixes	-0.0025	-0.034	0.000
enhancementFixes	-0.0022	0.06	0.015
patchFixes	-0.002	0.056	0.01
featureFixes	-0.0024	-0.178	0.000
enhancementReported	-0.0001	-0.208	0.000
patchReported	0.0001	0.041	0.024
p2-fixes	0.0005	0.176	0.004
p2-reported	0.0025	-0.067	0.038
p4-reported	0.0022	-0.31	0.000
p5-reported	0.0035	0.122	0.000
LineOperIRTolLines	-0.0491	-0.18	0.000
author	-0.0008	-0.037	0.103

Table 8.18: Open Office: Regression Model

Feature	**Unstand:**	**Standard:**	p
(Constant)	0.602		0.000
enhancementFixes	0.00027	0.391	0.000
patchFixes	0.004	0.155	0.000
featureReported	-0.0006	-0.141	0.000
p4-fixes	-0.0001	-0.083	0.035
p5-fixes	0.0024	0.65	0.000
p1-reported	-0.0001	-0.274	0.000
LineOperIRTolLines	0.026	0.057	0.102
author	-0.0007	-0.249	0.000

Table 8.19: Netbeans: Regression Model

Project	**Pearson**	**Spearman**	**MAE**	**RMSE**
Eclipse	0.59	0.308	0.046	0.061
Mozilla	0.57	0.361	0.045	0.057
Netbeans	0.65	0.623	0.041	0.056
Open Office	0.55	0.35	0.066	0.083

Table 8.20: Performance of the regression models: correlations are significant at $\alpha = 0.01$ level

Project	#of buggy files	Mean	Std. Dev.
Eclipse	10371	3.65	5.7
Mozilla	1585	9.77	18.8
Netbeans	10371	3.36	5.7
Open Office	1832	9.29	10.24

Table 8.21: Bug fixing rate per file: Mean value of Netbeans project significant at 0.01 level

Netbeans this factor has a comparatively high impact compared to the other features. In Eclipse, Mozilla, and Open Office, this factor contributes negatively to the model, while in Netbeans it contributes positively but not significantly. Hence, if the coefficient is negative (as in Eclipse, Mozilla, and Open Office) then an increased bug fixing activity (compared to new feature additions) will have a negative impact on the AUC – presumably as an increased overall bug fixing effort will increase general code stability and, however, change the relationship between the project level features and the prediction quality of a bug prediction model. In addition, more bug fixing effort will result in changing the underlying defect generation rules and consequently, the prediction quality will be dropped. The new defect generation rules have to be fed to the prediction models so that the models comply with new piece of information.

Our assumption for the cause of this relationship and the different influence in NetBeans is supported by the projects' data: here NetBeans has the smallest bug fixing rate per file (3.36) compared to the other three projects (Eclipse: 3.65, Mozilla: 9.77 and Open Office: 9.29) and the Netbean's mean bug fixing rate is significantly different from the other three projects at $\alpha = 0.05$ level. We compute the bug fixing rate per file by dividing the unique number of bugs fixed during the observed periods by the number of files that have at least one bug fixing activity during those periods. Table 8.21 shows the projects, number of buggy files, mean number of bugs fixed for a file and the standard deviation.

Also, it is worth to mention that the four regression models use different sets of features for their predictions. One reason for this could be that the observed projects are completely independent from each other in terms of authors, their workload, their experiences, development environment, *etc.* Therefore, the set of project features influences on the defect prediction quality varies from one project to another resulting in different regression models. Unfortunately, however, we have no firm theory as to why the predictors vary between projects.

Finally, we would like to point out that the somewhat moderate (but significant) correlations reported in Table 8.20 are no cause for concern. Indeed we embarked on this experiment with the goal of showing that such a prediction of an AUC is possible and that it produces promising results. We have achieved this goal given the results reported in the table. Obviously, there is room for improvement: a further exploration should consider non-linear regression models.

To conclude we found that we can predict the AUC of a defect prediction model with a decent accuracy (in terms of mean squared and absolute error). In addition, we found that the feature `author` has a consistent albeit small presence in all prediction models and may, therefore, have a universal applicability. Hence, we further explore this feature in the next sub-section.

8.2.5 Author fluctuation and bug fixing activities

The project feature `LineOpeIRTotLines` in the previous experiment encourages us to further investigate about the authors' contribution for bug fixing activities. This feature has a negative impact on the AUC (prediction quality measure) of Eclipse, Mozilla and Open office projects. However, this feature does not have any significant effect on the Netbeans model ($p = 10.2\%$). One could, therefore, hypothesize that in Eclipse, Mozilla and Open office projects most bugs are fixed by authors, who are not active in the training period but in the target period. To test this proposition we computed the fraction of bug-fixing work done by the authors, who are not in the training period but in the target period. Figure 8.22 graphs the result for one target period (the last month of the observed period; the others are omitted due to space considerations; but they look similar to this figure),where the x-axis represents the time difference between the target period and the two-month training window in months and the y-axis represents the fraction of bug fixing performed by authors, who are not active in the two-months training period, but in the one-month target period.

Results and Discussion

The Figure 8.22 clearly shows that in Eclipse and Mozilla an increasing proportion of bugs are fixed by those authors, who are not in the training period, and the fraction continuously increases the further we look back into past. In Open Office the fraction of work done by new authors drastically varies and is probably not meaningful due to a significantly smaller number of transactions (commits) per month.

For Netbeans the fraction of work done by authors, who are not in the two-months training period to fix bugs is initially very small and never rises above about 50% with a mean well below 40%. Further, the number for Netbeans is relatively constant indicating some stability in its developer base. Hence, mostly authors who active in the two-months training period is fixing bugs seems to be increasing the models prediction quality as those authors' behavior is well captured by the models.

The above observations encouraged us to further investigate the relationships between author fluctuation and bug fixing activity in periods of stability versus variability. To that end we identified tipping points from stable to variable periods in each of the projects and graphed the normalized change in number of authors and normalized change in bug fixing activity for the months preceding the onset of the variability and some months into the variability. Consider Eclipse (Figure 8.17) as an example: here the investigated months include "stable" months leading up to the tipping month of July 2003 and including the "variable" months until October 2004.

The value for the authors is computed as:

$$authchange_{month} = \frac{\#auth_{month} - \#auth_{month-1}}{\sum_{t \in months} |\#auth_t - \#auth_{t-1}|}$$

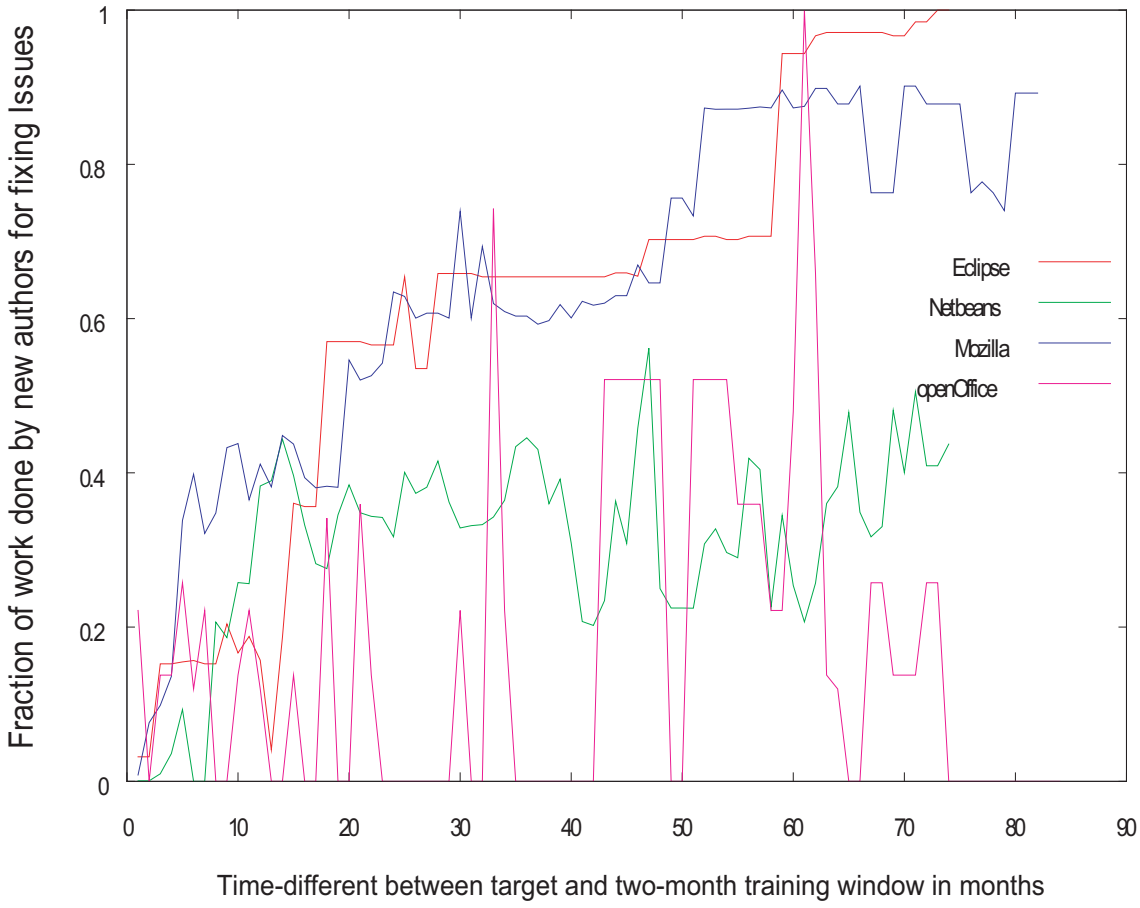


Figure 8.22: Work done by new authors to fix bugs

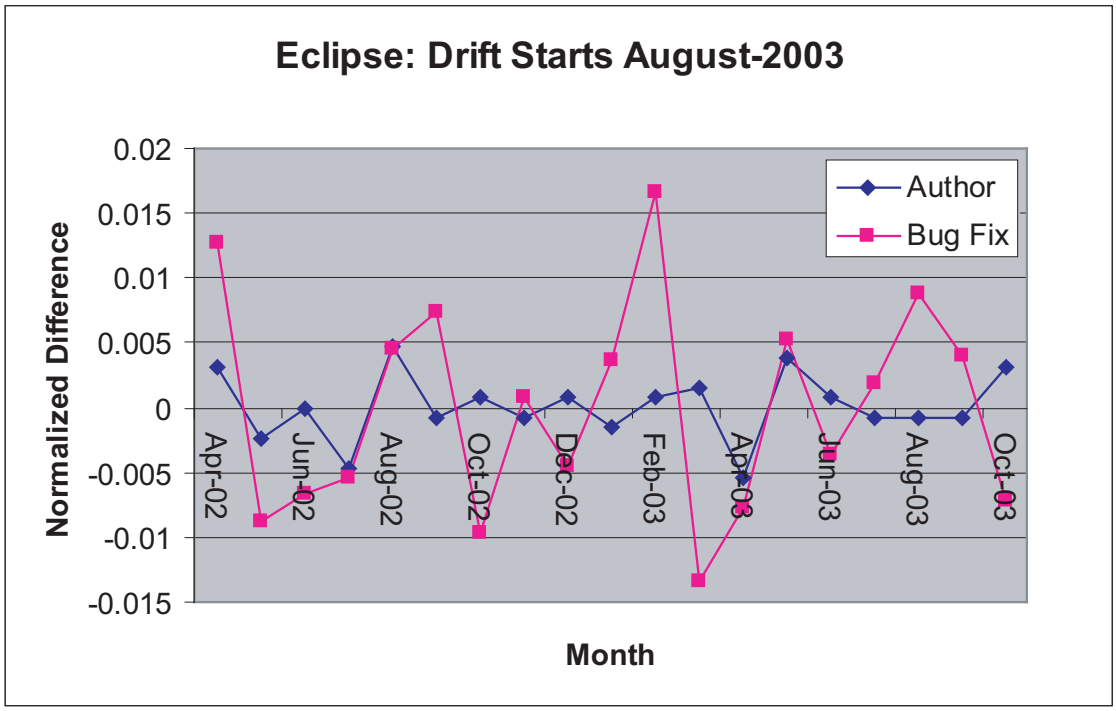


Figure 8.23: Eclipse: Tipping starts in July 2003

In words: the difference between the number of authors ($\#auth$) of the month ($\#auth_{month}$) and its preceding month ($\#auth_{month-1}$) normalized by the sum of the absolute differences of all the months considered in the graph. The value for changes in bug fixes is computed analogously. The rationale for the normalization is to make the figures somewhat comparable across different projects and time-frames.

Figures 8.23, 8.24,8.25, and 8.26 show a selection of the resulting figures, which are titled by the "tipping" month. All five figures show a relative drop in the number of authors in the "tipping" month mostly followed by an increase in authors during the drift. We also find that in most cases, the relative amount of work done for bug fixing increases massively in the "tipping" month. Unfortunately, none of these observations is unique to the tipping periods. Considering Eclipse (Figure 8.23), *e.g.*, we find that normalized author differential dips 3 times: on January 03, April 03, and preceding the drop down in prediction quality on July 03. The same can be said for the normalized bug differential. Hence, we cannot argue that these factors can be used exclusively to predict periods when the prediction quality starts declining, but together they can serve as a basis for developing such an early warning indicator.

Summarizing, we observe that increasing the number of authors editing the project has a negative impact on defect prediction quality. We also saw that more work done to fix bugs in relation to the other activities causes a reduction of the defect prediction quality. Further explorations

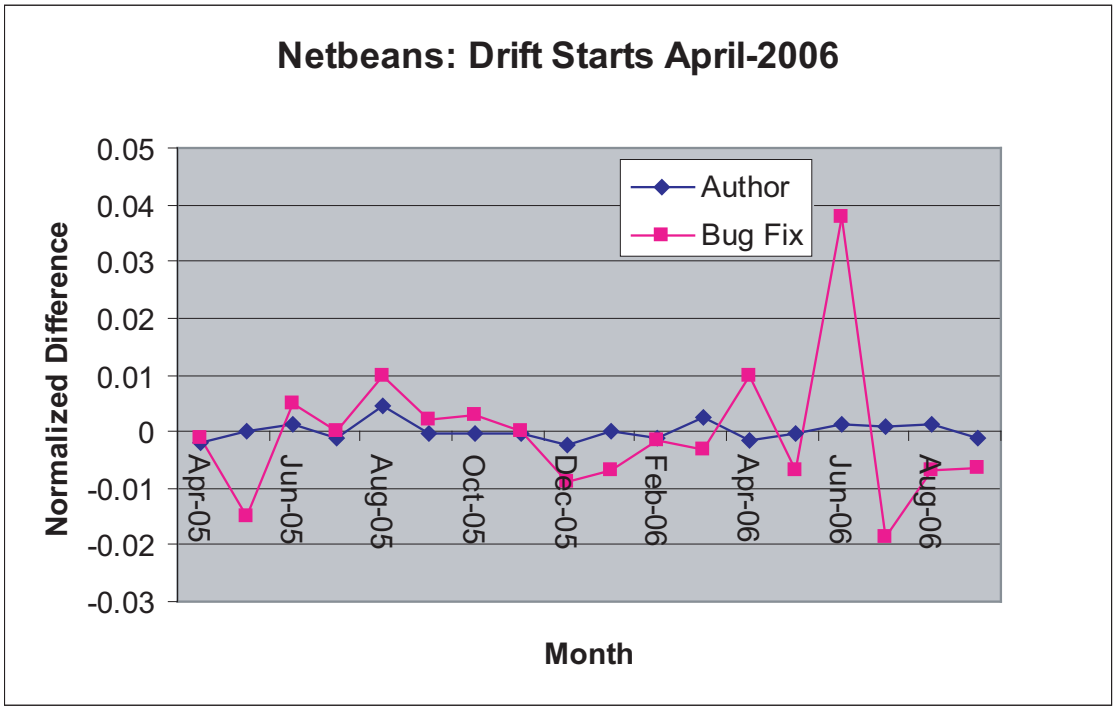


Figure 8.24: Netbeans: Tipping starts in April 2006

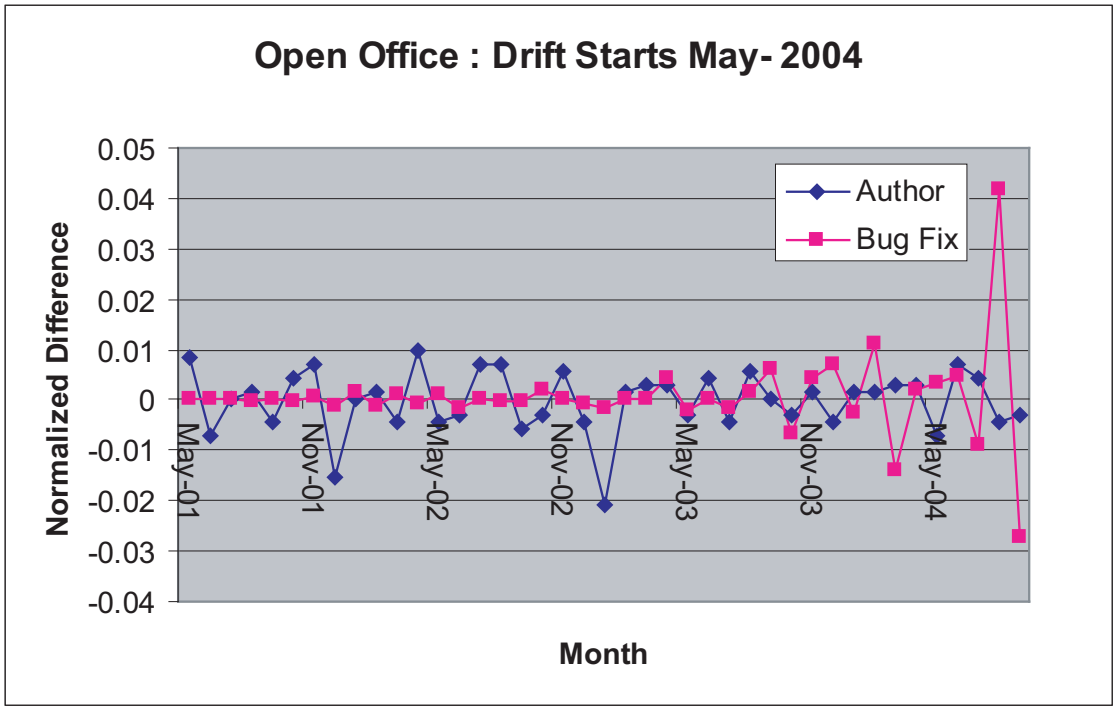


Figure 8.25: Open Office: Tipping starts in February 2004

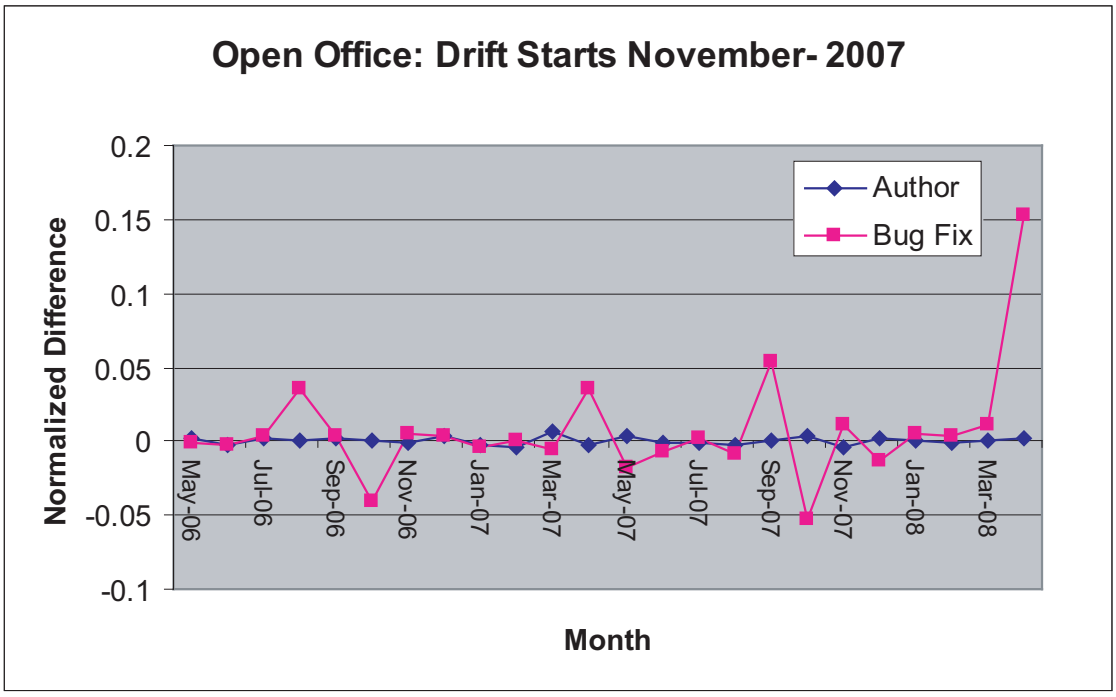


Figure 8.26: Open Office: Tipping starts in September 2007

Period	#of bugs fixed	Mean prio.	Std. Dev.
Instable	6976	2.22	0.487
Stable	8472	2.64	0.633

Table 8.22: Comparing mean priorities of stable and instable periods of Eclipse

Period	#of bugs fixed	Mean prio.	Std. Dev.
Instable	24278	2.39	0.79
Stable	23930	2.46	0.84

Table 8.23: Comparing mean priorities of stable and instable periods of Netbeans

indicated that when authors/developers, who are already present during the learning period are involving in fixing bugs helps increase prediction quality. These findings indicate that it is possible to uncover reasons, which influence defect prediction quality. Hence, we explore another possible factor – the mean priority of bugs fixed – in the next subsection.

8.2.6 Priority level of bugs influence for defect prediction quality

In the previous experiments we ascertained that we can indeed find periods of stability and change in our bug prediction models. Moreover, we found some indicators that can serve for upcoming change in prediction quality. In this section we continue our exploration for more indicators.

Specifically, when studying the regression models shown in Tables 8.16 – 8.19 it is striking that all models include at least one of the features pertaining to the priority of the bug-fixes. Hence, we decided to explore the mean of priority bug fixes and found it to be an indicator in both Eclipse and NetBeans. Specifically, we used the bug reports of Eclipse and Netbeans during the period from January 31, 2001 to May 31, 2007. In *Bugzilla*, and hence our data, the priority level of the bugs is discretized into five levels where the highest priority and the lowest priorities are assigned to 1 and 5 respectively and the other priority levels fall in-between. Having tested the normally of the priority values we compared the mean priority of the bugs fixed in the stable and instable periods as identified by the procedure explained in section 8.2.2. The results in Table 8.22 and 8.23 show that the difference between the mean priority of bugs fixed in the stable and unstable periods of two projects is statistically significant at $p = 0.001$ (using a t-test). Unfortunately, the found indicators do not generalize well to all projects. However this indicator can serve for possible change in prediction quality into a certain extent.

8.3 Concluding Discussion

In this chapter we investigated the notion of periods of stability and variability in data in software projects. Specifically, we were interested in such differing periods with respect to their impact on defect prediction algorithms. Using data from four open source projects we found that the quality of defect prediction approaches indeed varies significantly over time. We, furthermore, found

that the quality of the prediction clearly follows periods of stability and variability, indicating that *differing periods are indeed an important factor to consider* when investigating defect prediction. As a consequence, *the benefit of bug prediction in general must be seen as volatile over time and, therefore, should be used with caution*.

We observed that the number of authors editing the project is rising right before, or during periods of instability. This slightly reinforces the well-known software engineering lesson "adding manpower to a late software project makes it even later" [Brooks and Phillips, 1995]. We also saw a relationship between the changes of the proportion of work done to fix bugs and other activities and the changes in defect prediction quality. Unfortunately, both those correlations were not observed uniformly and can only serve as a start to elicit early warning indicators for changes in stability and, hence, the reduced quality of existing defect prediction models. Further, we found that authors, who are already active in the learning period of models are engaging in fixing bugs and are helping to improve defect prediction quality. Moreover, it was revealed that in some cases the priority level of bugs influences prediction quality. However, this finding is not consistent in all four projects.

During our experimentation we repeatedly asked ourselves if the causes behind the periods of stability and variability lie in *concept drift*. Concept drift is a notion from machine learning that refers to changes in the data generation process. Specifically, [Tsymbal, 2004] defines it as follows:

> "In the real world concepts are often not stable but change with time. ...Often these changes make the model built on old data inconsistent with the new data, and regular updating of the model is necessary. This problem is known as concept drift, ...drifts can occur suddenly (abruptly, instantaneously) or gradually"

Our phenomena exhibits strong attributes of *concept drift*: It shows periods of stability followed by periods of variability (= drift). Indeed, it was this "behavior" that inspired us to draw on some of previous work about prediction under *concept drift*and adapt it to our meta-prediction model [Vorburger and Bernstein, 2006]. In addition, we found some limited evidence for *concept drift*: the author fluctuations discussed in Section 8.2.5, *e.g.*, that indicate that the influx of new developers is associated with changes in variability. Obviously, these new authors may not be familiar with the norms of the projects and, hence, import new programming habits. This in turn may lead to the introduction of new bugs and changing the concept and generating the periods of variability we observe. Whilst this scenario seems plausible, we have no evidence for it and can at best hypothesize that *concept drift*is the cause for the changing periods of stability and variability. Finding strong evidence for *concept drift*must, therefore, be the focus of future investigations.

In this chapter we only scratches the surface of changing periods in software projects suggesting that *concept drift*may be one of its causes. Indeed, further investigations into the causes of these changes in software projects are needed – may they be rooted in *concept drift*or in a different cause. In the ideal case it would be possible to identify *the* influential factors that hold for software projects in general. Whatever the outcome of future investigations be, we can safely say that the notion of periods of stability and variability as well as probably that of *concept drift*seems to have

a profound influence in the empirical investigation of software evolution and needs to be taken seriously in any empirical software engineering study – in particular studies about predicting software bugs.

8.3.1 Threats to Validity

First, all the threats mentioned in the Section 3 are affecting these experiments too. Additionally following threats are influencing the outcome of these experiments.

Obviously, the generalizability of all our findings is curtailed by the limited number of projects considered. Whilst four projects is a decent size the generalizability of these findings needs to be investigated by looking at additional open and closed source projects. Furthermore, we find that the results are not as 'clean' as one would wish. Indeed, as the Figures 8.17, 8.18, 8.19, and 8.20 illustrate, the triangle shapes indicating periods of stability are sometimes difficult to discern even though we identified and highlighted them using an auto-detection method. We hope that further investigations may uncover the reasons for this seeming 'noise' in the data.

`Choice of Time Frames:`
We chose two-month windows as datasets for our prediction models. We do not insist that this window is the only or even correct one. We decided on 2 months because it was a time-frame that we found useful in Chapter 7 and is a release cycle that we observed in different projects. Often, a version/milestone is reached after 6-8 weeks. Obviously, software projects, just like any other project, often exhibit some form of entrainment (see [Ancona and Chong, 1996]). For future work it would be interesting to (i) assess the entrainment cycles and (ii) investigate the robustness of our results when narrowing the time windows to, e.g., days or weeks.

Part VI

Turning the Insights into Actionable Knowledge

9

Can prediction quality of models be predicted beforehand?[1]

Decision analysis is a general phrase describing the broad application of modeling and simulation techniques for improving decision-making. Decision support tools make it possible for software managers to apply decision analysis techniques throughout their organization to problems ranging from simple projects to enterprise-wide strategic plans. Moreover, decision support tools can help the project managers anticipate future risks and opportunities, so that they can act rather than react. To our knowledge there is no decision procedure or a tool that support software managers to decide when and when not it is beneficial for them to apply prediction models. This is very crucial information for the managers since in Chapter 8 we showed that defect prediction quality varies over time such that there exists a period of stability and change.

Particularly, in the Subsection 8.2.4 we found that a prediction model can be trained to predict the performance of bug prediction models. Our approach essentially devises a prediction model of a prediction model, which we will call 'meta-prediction model' in the following. The main remaining question is, if such a meta-model can be used within a decision procedure for software project managers. In this section, we address this issue and present such a decision procedure that relies on these meta-prediction models.

Consider the meta-prediction models learned in Subsection 8.2.4 and shown in Tables 8.16–8.19. As we argued these meta-prediction models can predict the AUC of the bug prediction model at any given time period using the project features. Assuming that these predictions are good – and we showed in Table 8.20 that they are at least decent – then it would seem to be natural to use these predictions as a decision measure about the expected quality of bug prediction methods.

Specifically, a software manager hoping to attain a reliable indication for the location and quantity of bugs should only use bug prediction methods when they can be expected to have a certain prediction quality. A good indicator for the expected quality of a bug prediction method that we have is the value generated by the meta-prediction model. Hence, *she should only use the bug prediction method when the meta-prediction model predicts a AUC above a certain threshold*. We have already shown in Section 8.2.2 that the AUC > 0.65 is a "decent" result when compared to the award-winning BugCache.

[1] This section was published in [Ekanayake et al., 2011]

To show that this method works, Figures 9.1– 9.4 graph the average AUC of all actual predictions gained from the bug prediction models that were predicted to have an AUC above the threshold by the meta-prediction model on the y-axis whilst varying the threshold on the x-axis. Note that whenever the meta-prediction model makes a bad prediction it will result in an AUC below the threshold.

The figures show that raising the thresholds will eventually lead to better predictions. The figures also show some prediction quality instabilities with a rising threshold, and in 3 of the 4 cases a collapse of prediction quality at the very end. Whilst this is disappointing at first, it becomes almost logical when considering the number of data-points that decreases with the rising threshold. In other words; the further right in the figure one looks the less actual predictions are used to compute the average. This has two consequences; first, at the very end only one model is over the threshold and the "average" is really the prediction quality of that model. Secondly, as the number of data points included in the average decreases, it also becomes increasingly influenced by single misjudgments. Hence, the instability, which initially is quite disappointing becomes understandable.

As a consequence, we can conclude that our proposed approach actually works reasonably good for all projects. Indeed, choosing thresholds (> 0.8 for Eclipse and Mozilla; > 0.6 for OpenOffice;) will assure a manager that her model will obtain the minimum required prediction quality (AUC 0.65). However, it does not imply that the model's prediction quality cannot exceed that limit. It can vary in the rage of 0.65, which is the minimum in our case to 1.0. For Netbeans, the threshold is > 0.7 and it gives approximately 0.61-0.62 minimum prediction quality. The prediction quality looks rather low. It should be noted that in Netbeans, we explored 93 subcomponents for this experiment. These subcomponents have been developed under different development environments (different authors, tools, etc.). Therefore, we can expect a large variability in Netbeans-data and consequently, the models' prediction quality could be dropped. However, this experiment is not designed for testing this proposition and that is a venue for future research.

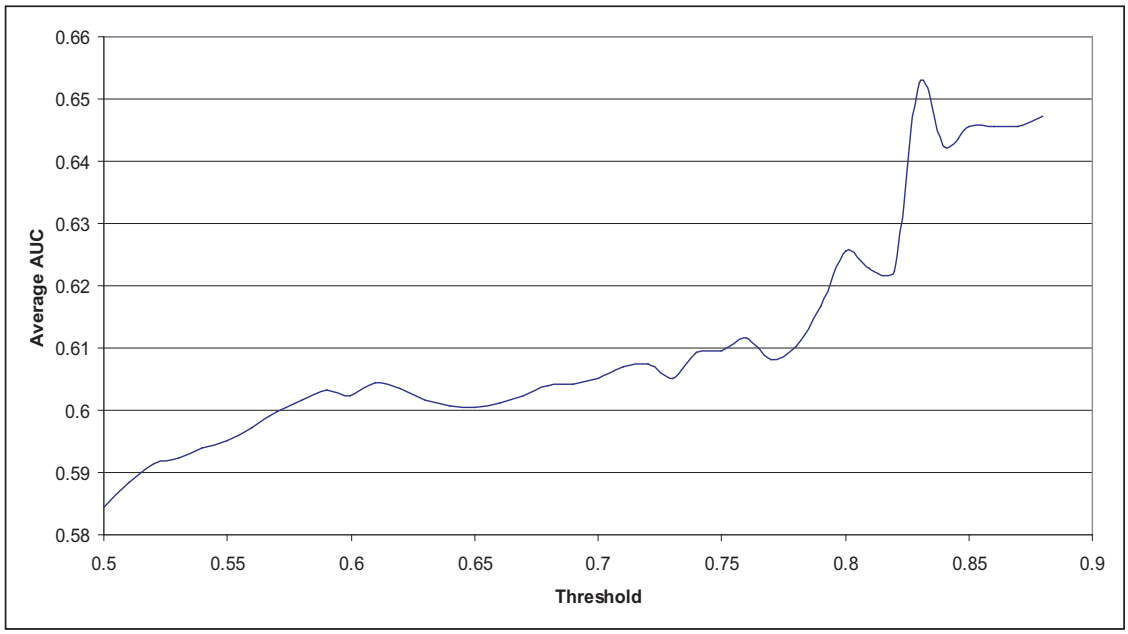


Figure 9.1: Eclipse: Graphs estimate the actual AUC based on the predicted AUC by the linear models

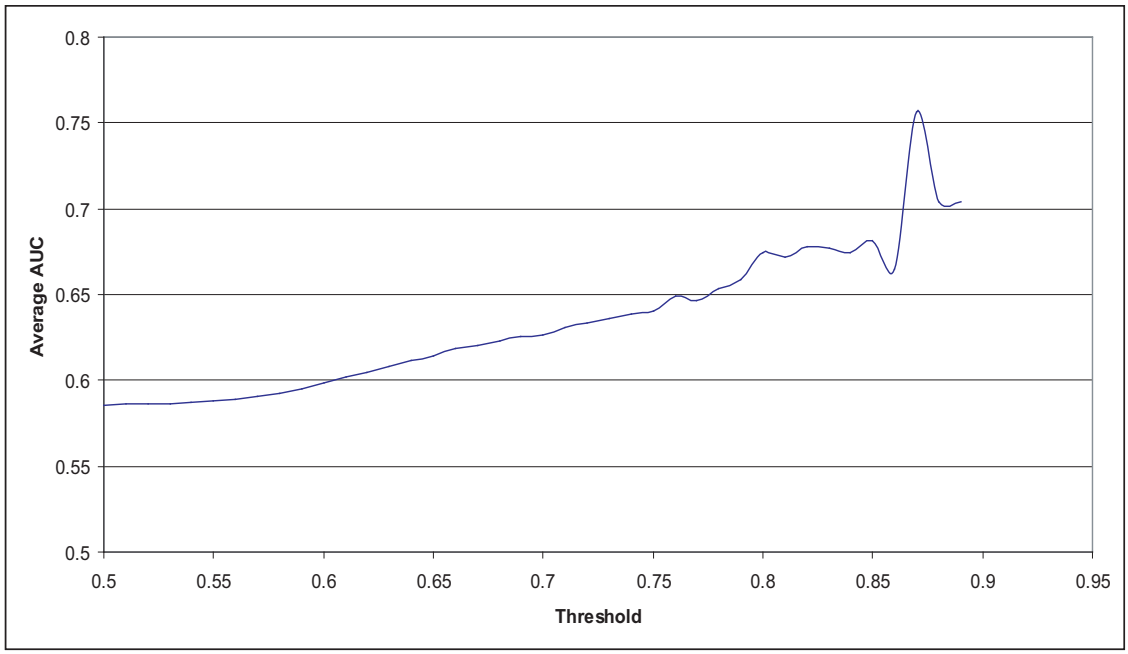


Figure 9.2: Mozilla: Graphs estimate the actual AUC based on the predicted AUC by the linear models

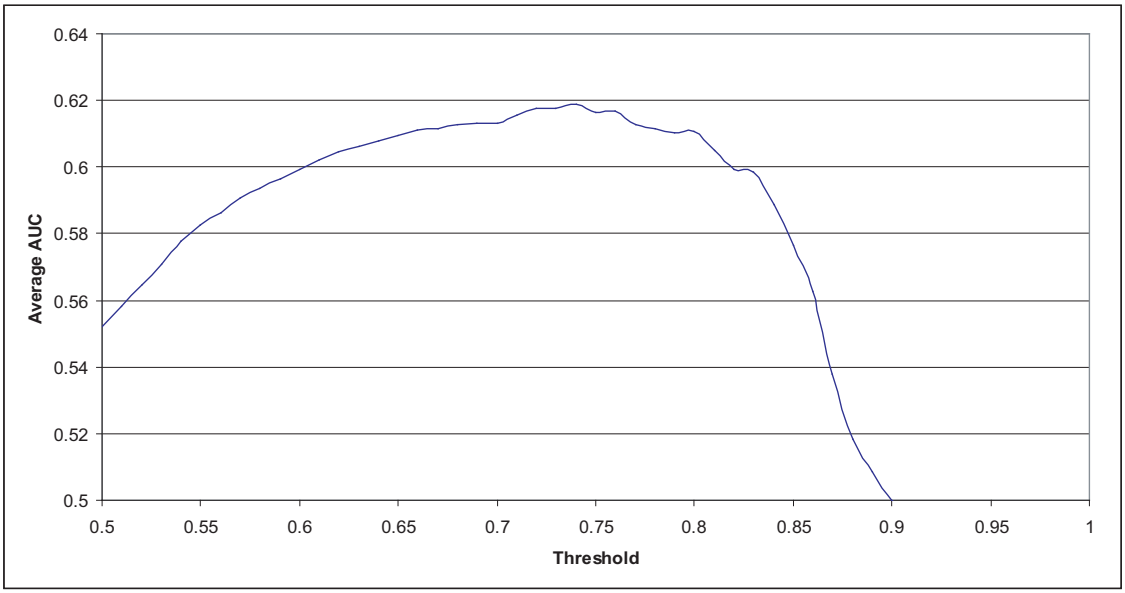


Figure 9.3: Netbeans: Graphs estimate the actual AUC based on the predicted AUC by the linear models

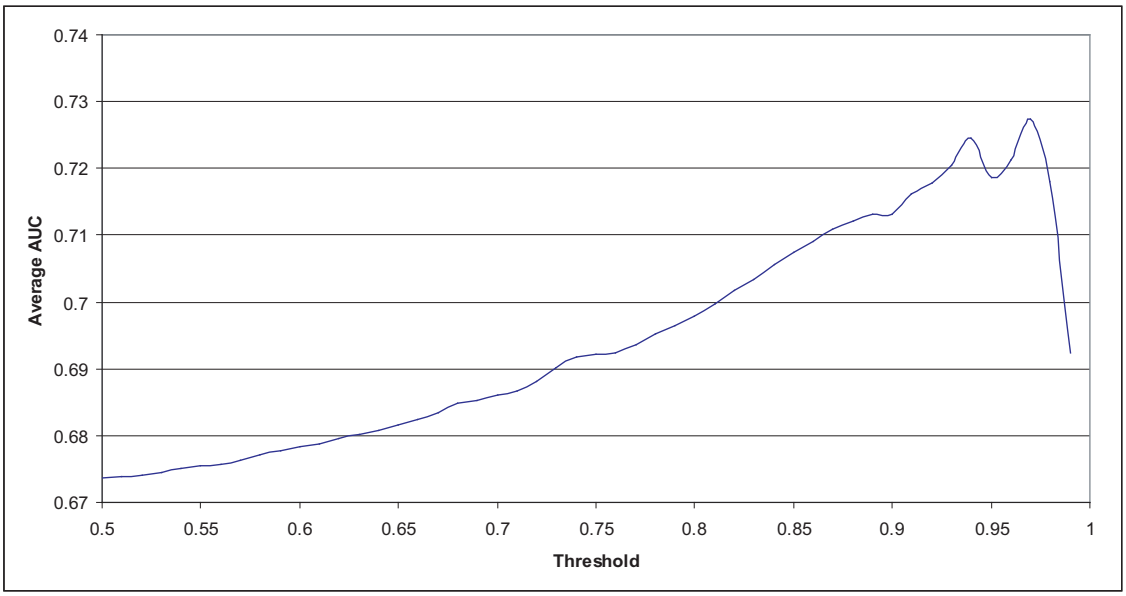


Figure 9.4: Open Office: Graphs estimate the actual AUC based on the predicted AUC by the linear models

Part VII

Summarizing Discussion, Future Works, and Conclusion

10

Summarizing Discussion

Software systems play an important role in business and other sectors such as health, military, transport *etc.* The growth in computer use and computer hardware capabilities has placed demands of increasing magnitude and complexity on computer software. As the sizes of software projects have increased, software development processes based on individual programmers have given way to processes based on small teams and, in turn, small teams have given way to larger teams and so on. Higher scaling of software development processes by merely increasing team sizes reaches limits on effective project management and resource availability. Today's users of software demand software applications of greater size and complexity than before. Advances in computer hardware capabilities are more than adequate to match the demands of users; however, software is not. The challenge is to develop software with attendant methodologies and technologies that meet user demands and that improve software quality and productivity.

Improving software quality and productivity is a major challenge faced by the developers. Improving software quality is basic to how well software meets the requirements and expectations of the users. It also means ensuring that software is adequate, reliable, and efficient. Improving productivity means favorably increasing the ratio between the resources required to develop software and the size and complexity of the developed software. Hence, a development process associated with efficient handling of concurrent development and fixing bugs is needed for improving the software quality and productivity. Unfortunately, software development and maintenance is an error-prone, time-consuming and complex activity. Experience has revealed that many software development efforts falter because the management of these projects fall into several common traps. Consequently, many tools have been developed in last decades to help software developers and engineers to improve the software quality. Among them, prediction models (bugs, cost, refactoring *etc.*), bug tracking systems, version control systems, integrated development environments are very valuable tools.

However, the reliability of the bug prediction models is very important since the fixing effort is planned according to the prediction. In the last decades several bug prediction models have been developed. Specifically, those models were trained using the software engineering process data extracted from version control systems (VCSs) and bug tracking systems (BTSs). Most of the bug prediction models shared a common methodology: first, features or variables were extracted from a certain time period to train models and then those models were evaluated using the same

set of features that were extracted from another one or few time points. This evaluation method implicitly assumes that the underline data distribution is relatively stable over time, which is not necessarily be valid. Therefore, those models may not be well generalized over all points in time. To investigate this issue in details we first wanted to understand whether real-time information influences prediction quality. To counter this issue we defined our first Research Question:

Q1: Do time-based sampling techniques influence defect and cost prediction quality?

In order to investigate the above question we defined two time-based data sampling techniques:

Temporal Sampling Technique: that collects prior to prediction periods – target periods – for training models. This sampling technique assumes that information prior to the prediction periods is a better predictor.
Random Sampling Technique: In contrast to the above sampling method, this sampling method does not take the time factor into consideration and collect data from before and after the target period.

Next, we defined two hypotheses to address the above question.

H 1.1: *In the defect prediction domain, models trained on data collected from the temporal sampling technique are better than models trained on data collected from the random sampling technique.*

H 1.2: *In the cost estimation domain, models trained on data collected from the temporal sampling technique are better than the models trained on data collected from the random sampling technique.*

For this experiment we used software data from Eclipse and Netbeans OSS projects and cost estimation data from ISBSG Repository Data Release-9. We used WEKA APIs inside a Java Development Environment (JDE) to implement the models. We first implemented the temporal sampling method. In the temporal sampling we always trained prediction models from the data collected prior to the prediction period – target period (see Section 6.1.1 for dataset construction). We started the experiment with training models from software data for predicting defects. We predicted defects for each target of the observation period using the data prior to that target. The models' prediction quality were measured using the Person correlation coefficient between the true and predicted values, MAE and RMSE. The same experiment was conducted with cost estimation data. We then conducted the random sampling experiment with software and cost estimation data. In the random sampling experiment the data was collected randomly before and after the target period but, not from the target period. We then compared the prediction quality between these two sampling techniques (see Tables 10.1 and 10.2 for prediction quality in defect prediction and cost estimation respectively).

	Temporal	Random	
		Mean	Median
Correlation	0.21	0.19	0.18
MAE	0.75	0.78	0.77
RMSE	0.92	0.91	0.9

Table 10.1: Comparison between random and temporal sampling in defect prediction (extracted from Tables 6.4–6.5)

	Temporal	Random
Correlation	0.52	0.51
MAE	5758.7	5934.5
RMSE	12371.0	10248.02

Table 10.2: Comparison between random and temporal sampling in cost prediction (extracted from Table 6.10)

These results concluded that in general, temporal sampling is better than random sampling in defect prediction since the correlation of temporal sampling is significantly higher than the random sampling method. Furthermore, the MAE of temporal sampling is significantly lower than the random sampling. However, the RMSE values of both methods have no significant different. The observed software projects evolved over time and the past changes influence for future changes. Therefore, past information is a better predictor of future events. This implies that the time-based information collection for predicting defects is essential. Furthermore, random sampling may lose vital information and consequently, the prediction quality is dropped down.

It is interesting to know that there is no significant difference between these two methods on cost estimation. The reason behind this observation is the cost estimation data contains final cost. It does not contain the cost for each development stage of a project as in software data. Therefore, data collection based on time for training models is not necessary. Moreover, this result supports the finding by Jay Forrester nearly forty years ago [Porter, 1962] dynamics-based techniques explicitly acknowledge that software project effort or cost factors change over the duration of the system development; that is, they are dynamic rather than static over time.

As a summary, based on the outcome of this experiment we can support the first hypothesis (H 1.1), which is temporal sampling is better than random sampling in the case of defect prediction. But, we did not have enough evidences to support the second hypothesis (H 1.2). More importantly, the outcome advices us to use all information prior to the prediction period for better defect prediction.

In our previous work we uncovered temporal information *i.e.* prior information is better defect predictors. Further, we uncovered that randomly selected information has negative impact on prediction quality. We make use of these finding for the next experiment.

As we mentioned earlier the software history is a valuable source of information and hence, we can extract many features – static and temporal – or variables from it. Also, in the field of machine learning many robust learning algorithms have been developed. Consequently, it is not an easy

task for software developers to select the most appropriate feature set and an algorithm to train a defect prediction model. Hence, this is an interesting question to address and we defined the following Research Question:

Q2: Which type of features – temporal or static – and, models – linear or non-linear – are improving the prediction quality?

To address the above research question we defined two hypotheses as follows:

H 2.1: *Models trained from temporal features are more predictive than models trained from static features.*

H 2.2: *Non-linear models are more precise on predicting defect than linear models.*

For this experiment we used software data from six plug-ins of Eclipse OSS project. We first constructed features both temporal and static for training the models from the historical information of those six plug-ins. The static features are constructed from the entire observation period and the temporal features are computed from the scratch of the observation period such as 1,2,3, or 5 months time frames from the prediction period. We used J48 decision tree learning algorithm and M5P regression tree algorithm to train prediction models. Moreover, the models were trained with different base-sets of features either using static features whatsoever or using the temporal features for different window sizes of 1, 2, 3, and 5 months. The models were evaluated using separate test sets.

The experimental results (see Tables 7.3 and 7.5) showed that the temporal features are dominating in predicting defects and hence, we experimentally, proved the the first hypothesis (H 2.1). More importantly, the models choose temporal features when ever possible and those features are appeared at the root or at very higher levels . The accuracy measures (AUC) clearly showed the temporal features better than static features. Moreover, we showed that more recent temporal data is more useful than older one. Hence, we can argue that modules with bugs are likely to have bugs in later versions, but over longer periods of time those bugs could be fixed. In other words: Bugs are likely to survive some versions, but are fixed after some. The same fact was uncovered by several researches [Hassan and Holt, 2005],[Kim et al., 2007], but using different techniques. Furthermore, we compared our models with the models developed for the same task – predicting defects on Eclipse – and found that our significant-features model is better than those models (see Table 7.7).

In order to prove our second hypothesis (H 2.2) – non-linear models are better than linear models in predicting defects – we trained linear regression models with the same set of features and the prediction quality of those models were compared with the non-linear models (see Table7.8). The results showed that the non-linear models are better than linear models in predicting defects.

Finally, we showed that our bug prediction models predict the upcoming bugs with a decent accuracy (see Table 7.9).

In a summary, in this Chapter we uncovered the temporal features and non-linear models are more appropriate in training defect prediction models. The temporal features are instances of static features and they explain the evolution of static features. Further, this is another evidence that the time-based information is more important in the defect prediction.

It is crucial for a software manager to know whether she can rely on a bug prediction model or not. A wrong prediction of the number or the location of future bugs can lead to problems in the achievement of a project's goals. Several defect prediction models have been developed in past decades. Most of them shared similar procedure to train and evaluate the models. First, the models were trained on data collected at one time period and they were evaluated on data collected at another one or few points in time. This evaluation implicitly assumed that the evolution of the software projects are relatively stable over time. But, this assumption is not necessarily valid. Hence, generalization of such models is difficult. Also, the past studies showed that evolving systems can be subjected to concept drifts and the models trained on data before the concept change may not be valid with new concepts. The software systems are also evolving over time and hence, the existence of concept drift in software projects can not be ruled out. Therefore, it is important to investigate the possible variability in prediction quality and consequently, we defined our third Research Question:

Q3: Is the prediction quality constant in every time period?

To investigate this question we define three hypotheses:

H 3.1: *Defect prediction quality varies over time.*

H 3.2: *There exists periods of stability and change in prediction quality.*

H 3.3: *There exists features that influence for prediction quality.*

We explored four OSS projects – Eclipse, Netbeans, Open Office and Mozilla – to investigate the above research question and the hypotheses. In this investigation we used J48 decision tree models and linear regression models to train prediction models. First, we verified the existence of variability in a bug prediction model's accuracy over time both visually and statistically (H 3.1). This experiment results showed that the prediction quality varies over time: when predicting defects in different target by using the same model and predicting the defects in same target by using different models (see Figures 8.1–8.4 and Figures 8.9–8.12). Second, we showed that there exist periods of stability and variability of prediction quality (H 3.2). We visualized the periods, where the prediction quality is decent and can be used for decision making, and the periods, where the prediction quality is not decent enough for making decisions (see Figures 8.17–8.20). Furthermore, we explored the reasons for such a high variability over time, which includes periods of stability and variability of prediction quality (H 3.3). Specifically, we observed that a change in the number of authors editing a file and the number of defects fixed by them consistently influences

the prediction quality. Further exploration indicated that when authors/developers, who already presented during the learning period are involving in bug fixing activities helps in increasing prediction quality. More importantly, we showed that the prediction quality can be predicted using project features. Our findings suggested that the software managers should be aware of the periods of stability and variability of prediction quality before applying their prediction models.

Summarizing, this investigation uncovered that possible variability in prediction quality. Further, we uncovered the features that change in the number of authors editing a file and the number of bugs fixed by them for influencing the prediction quality and hence, the software managers can understand the upcoming variability in defect prediction quality by looking at these features. Finally, we repeatedly asked by ourselves that this variability in prediction quality over time is due to the concept drift. But, we were not able to find strong evidences to support this hypothesis. However, to our knowledge, we were the first to discussed about the possible concept drifts in software projects.

In the previous investigation we found that the prediction quality varies over time. This causes an uncertainty for software managers when applying their bug prediction models. Specifically, the software managers should know whether the prediction quality of their models is decent enough for making decisions based on the predictions. Hence, we defined our last Research Question of this thesis:

Q4: Can the prediction quality of a model be estimated in advance?

Further, we defined the following hypothesis to investigate the above research question:

H 4.1: *A decision procedure can be defined to measure the quality of prediction models in advance.*

In order to define the decision procedure that measures the quality of prediction models in advance we used meta-prediction models learned in Subsection 8.2.4 and shown in Tables 8.16, 8.17, 8.18, and 8.19. We showed that these models can predict the AUC of the bug prediction models with decent accuracy at any time period (see Table 8.20) and used these predictions as a decision measure about the expected quality of bug prediction methods. In order to show that this method works, we graphed (see Figures 9.1–9.4) the average AUC of all actual predictions gained from the bug prediction models that were predicted to have an AUC above the threshold by the meta-prediction model. This decision procedure can be used to estimate the prediction quality of the models such that whenever the meta-prediction model makes a bad prediction – an AUC below the threshold – then the project manages can avoid the predictions by the models.

Summarizing, we found this approach actually works quite well for all projects. According to Figures 9.1–9.4, choosing thresholds (> 0.8 for Eclipse and Mozilla; > 0.6 for OpenOffice;) will assure a manager that her model will obtain the minimum required prediction quality (AUC 0.65). However, it does not imply that the model's prediction quality cannot exceed that limit. It can vary in the rage of 0.65, which is the minimum in our case, to 1.0. For Netbeans, the threshold

is > 0.7 and it gives approximately 0.61-0.62 minimum prediction quality. The prediction quality looks rather low. Please note that in Netbeans, we explored 93 subcomponents for this experiment. These subcomponents have been developed under different development environments (different authors, tools, etc.). Therefore, we can expect large variability in Netbeans-data and consequently, the models' prediction quality could be dropped down. However, this experiment is not design for testing this proposition and that is a venue for future research. Furthermore, this is a tool that we found by turning all the insights, which we gathered during this experiments into actionable knowledge.

11

Future Work

We presented interesting results in this thesis. Also, we have discussed possible limitations of each experiment result. Most of the limitations are pointing to future research. Hence, in this section we discuss most appropriate future works identified from this thesis.

11.1 Extending Experiments for Closed Source Projects

In this work we explored only open source software projects. Hence, it is interesting to analyze the same hypotheses using closed source projects and generalized the findings. As we explored four reasonably large open source projects we do not assume changes in the findings. However, we must admit the fact that getting closed source project data requires much effort. But, we strongly recommend to extend our work for closed source projects.

11.2 Data Quality

Data quality is an important issue in empirical software engineering research. Hence, we ensured the data quality by applying the approach revealed by Zimmermann [Zimmermann et al., 2007] . However, we identified few limitations of this approach. Therefore, in our future works we will use other approaches presented by Bachman *et al.* [Bachmann and Bernstein, 2009], Śliwerski *et al.* [Śliwerski et al., 2005] *etc.* to enhance the data quality. But, they also adapted slightly similar techniques as described by Zimmermann. Therefore, it is reasonable to argue that our inferred conclusions do not change by adapting the above data quality enhancement approaches.

11.3 Influence of Data Mining Tools and Algorithms

For all of our experiments we used WEKA implemented learning algorithms (J48, M5P, linear regression algorithm). Therefore, it is important to use other data mining tools such as Rapid-

Miner[1], R-package[2], Matlab[3] *etc.* to avoid the dependency on mining tools. Furthermore, we mostly used J48 decision tree learner and it is interesting to observe the outcome from other non-linear algorithms such as Support Vector Machine (SVM).

11.4 Biased on Process Metrics

In defect prediction we used only product metrics to train the prediction models. One could therefore argue that the outcome of those experiments are biased for process metrics. But, essentially the product metrics are influenced by process metrics. For instance, the workload of an author influences for the source code. Therefore, the product metrics and the process metrics are highly correlated. Usually, bug prevention methods are more important than bug detection methods. Hence, the process metrics are valuable assets for studying the bug introduction processes. However, we propose to investigate the same hypotheses using product metrics to avoid the dependency.

11.5 Concept drift in Software Projects

During the experiments in Chapter 8 we were curious about possible concept drifts in software projects. Specially, we are interested about the causes behind the period of stability and variability of defect prediction quality. We believe that concept drift is one possible cause for such significant variability. However, in this study we were not able to find strong evidences for such phenomenon. Hence, further investigation is essential for a conclusion and we strongly encourage researchers to focus on this matter.

[1] http://rapid-i.com/
[2] http://www.r-project.org/
[3] http://www.mathworks.com/products/matlab/

12

General Conclusion

In this thesis we first highlighted the drawbacks of existing defect prediction models and then presented time-based reasoning techniques to overcome these drawbacks. We empirically proved that those techniques have significant influence on improving defect prediction quality that ultimately contributes for enhancing the software quality and reliability. For this, we explored four open source projects and one cost estimation dataset to investigate the impact of time-based reasoning on defect prediction quality and cost estimation.

We discovered that time-based sampling is better than random sampling, which ignores the time factor and some important information, in predicting defects. Contrary to the above finding, those two sampling techniques have similar impact on cost estimation in software projects. These observations conclude that real-time information is more appropriate to predict events of chronological evolving process. Furthermore, it reveals that every information is important to model the relationship between the defects and the project features. Further investigations uncovered that temporal features and non-linear models enhance prediction quality. This implies that the relationship between the defects and projects features are not linear and hence, in future, the developers must use non-linear models for training defect prediction models. More important, we revealed that the defect prediction quality varies both predicting the same target by using several models and predicting different targets using the same model. Specifically, this finding suggested that testing a model on one or very few targets is not sufficient to generalize the model. We discovered several project features such as number of authors editing a file and the number of defects fixed by them are influencing the defect prediction quality and hence, we conclude that the prediction quality of a model can be predicted in advance. We turned the above finding into an actionable knowledge by developing a decision procedure that helps project managers to evaluate quality of their prediction models in advance. We strongly suggest for developers to apply this decision procedure before making any decisions based on their bug prediction models. In addition, we highly encouraged researches to investigate about the concept drift of software projects since we believe that variability in prediction quality may be caused by the change in underlying data distribution.

In brief, we empirically showed that our time-based reasoning techniques improve the prediction quality. Since the prediction quality of models significantly varies over the time we highly

recommend to use our decision procedure, which estimates the models' prediction quality in advance.

Part VIII

Bibliography

Bibliography

Ancona, D. and Chong, C. L. (1996). Entrainment: Pace, Cycle, and Rhythm in Organizational Behavior. In *Research in Organizational Behavior*, volume 18, pages 251–284. JAI Press.

Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., and Guéhéneuc, Y.-G. (2008a). Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research (CASCON)*, pages 304–318. ACM.

Antoniol, G., Gall, H., Pinzger, M., and Penta, M. D. (2008b). Mozilla: Closing the circle.

Anvik, J. (2006). Automating bug report assignment. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 937–940, New York, NY, USA. ACM.

Aversano, L., Cerulo, L., and Del Grosso, C. (2007). Learning from bug-introducing changes to prevent fault prone code. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, IWPSE '07, pages 19–26, New York, NY, USA. ACM.

Ayari, K., Meshkinfam, P., Antoniol, G., and Di Penta, M. (2007). Threats on building models from cvs and bugzilla repositories: the mozilla case study. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, CASCON '07, pages 215–228, New York, NY, USA. ACM.

Bachmann, A. and Bernstein, A. (2009). Data Retrieval, Processing and Linking for Software Process Data Analysis. Technical Report IFI-2009.0003, University of Zurich, Department of Informatics.

Bachmann, A., Bird, C., Rahman, F., Devanbu, P., and Bernstein, A. (2010). The Missing Links: Bugs and Bug-fix Commits. In *ACM SIGSOFT / FSE '10: Proceedings of the eighteenth International Symposium on the Foundations of Software Engineering*, page to appear, Santa Fe, USA.

Baird, B. (1989). *Managerial Decisions Under Uncertainty*. John Wiley Sons.

Baker, M. J. and Eick, S. G. (1994). Visualizing software systems. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 59–67, Los Alamitos, CA, USA. IEEE Computer Society Press.

Bakota, T., Ferenc, R., Gyimothy, T., Riva, C., and Xu, J. (2006). Towards portable metrics-based models for software maintenance problems. In *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*, pages 483 –486.

Basili, V. R., Briand, L. C., and Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22:751–761.

Bernstein, A., Ekanayake, J., and Pinzger, M. (2007). Improving defect prediction using temporal features and non linear models. In *IWPSE '07: Ninth international workshop on Principles of software evolution*, pages 11–18, New York, NY, USA. ACM.

Bettenburg, N., Just, S., Schröter, A., Weiß, C., Premraj, R., and Zimmermann, T. (2007). Quality of bug reports in eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, eclipse '07, pages 21–25, New York, NY, USA. ACM.

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. (2008). What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 308–318, New York, NY, USA. ACM.

Binkley, A. B. and Schach, S. R. (1998). Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pages 452–455, Washington, DC, USA. IEEE Computer Society.

Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., and Devanbu, P. (2009). Fair and Balanced?: Bias in Bug-fix Datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering (ESEC/FSE)*, pages 121–130, New York, NY, USA. ACM.

Blum, A. L. and Langley, P. (1997). Selection of relevant features and examples in machine learning. *Artif. Intell.*, 97:245–271.

Boehm, B. W. (1981). *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition.

Bouktif, S. (2004). Improving rule set based software quality prediction: A genetic algorithm-based approach. *Journal of Object Technology*, 3(4):227–241. Proceedings of the TOOLS USA 2003 Conference, 30 September - 01 October 2003 — Santa Monica, CA.

Bouktif, S., Sahraoui, H., and Antoniol, G. (2006). Simulated annealing for improving software quality prediction. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, GECCO '06, pages 1893–1900, New York, NY, USA. ACM.

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA. new edition [**?**]?

Brooks, F. P. and Phillips, F. (1995). *The mythical man-month: essays on software engineering*. Addison-Wesley Reading, MA.

Canfora, G. and Cerulo, L. (2005). How software repositories can help in resolving a new change request. In *In Workshop on Empirical Studies in Reverse Engineering*.

Catal, C. and Diri, B. (2009). Review: A systematic review of software fault prediction studies. *Expert Syst. Appl.*, 36:7346–7354.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20:476–493.

Cortes, C. and Vapnik, V. (1995). Support vector networks. *Machine Learning*, 20(3):273–297.

Cubranic, D. (2004). Automatic bug triage using text categorization. In *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering Knowledge Engineering*, pages 92–97. KSI Press.

Cun, Y. L., Denker, J. S., and Solla, S. A. (1990). Advances in neural information processing systems 2. chapter Optimal brain damage, pages 598–605. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Demeyer, S., Ducasse, S., and Nierstrasz, O. (2000). Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 166–177, New York, NY, USA. ACM.

Denaro, G., Morasca, S., and Pezzè, M. (2002). Deriving models of software fault-proneness. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, SEKE '02, pages 361–368, New York, NY, USA. ACM.

Ekanayake, J., Tappolet, J., Gall, H., and Bernstein, A. (2011). Time variance and defect prediction in software projects. *Empirical Software Engineering*, pages 1–42. 10.1007/s10664-011-9180-x.

Ekanayake, J., Tappolet, J., Gall, H. C., and Bernstein, A. (2009). Tracking Concept Drift of Software Projects Using Defect Prediction Quality. In *Proceedings of the 6th IEEE Working Conference on Mining Software Repositories*. IEEE Computer Society. to appear.

Fenton, N. E. and Neil, M. (1999). A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25(5):675–689.

Fischer, M., Pinzger, M., and Gall, H. (2003a). Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 90–, Washington, DC, USA. IEEE Computer Society.

Fischer, M., Pinzger, M., and Gall, H. (2003b). Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 23–, Washington, DC, USA. IEEE Computer Society.

Gegick, M., Rotella, P., and Xie, T. (2010). Identifying security bug reports via text mining: An industrial case study. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 11 –20.

German, D. (2004). Mining cvs repositories, the softchange experience. In *MSR 04: Proceeding of the 1st International Workshop on Mining Software Repositories*, pages 17–21, Edinburgh, Scotland,UK. ACM.

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

Gray, A. R. and MacDonell, S. G. (1997). A comparison of techniques for developing predictive models of software metrics. *Information and Software Technology*, 39(6):425 – 437.

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182.

Gyimothy, T., Ferenc, R., and Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng.*, 31:897–910.

Hall, M. A. and Holmes, G. (2003). Benchmarking attribute selection techniques for discrete class data mining. *IEEE Trans. on Knowl. and Data Eng.*, 15:1437–1447.

Halstead, M. (1977). *Elements of Software Science*. Elsevier, New York.

Hassan, A. E. (2009). Predicting faults using the complexity of code changes. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 78–88, Washington, DC, USA. IEEE Computer Society.

Hassan, A. E. and Holt, R. C. (2005). The top ten list: Dynamic fault prediction. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 263–272, Washington, DC, USA. IEEE Computer Society.

Helmer, O. (1966). *Social Technology*. Basic Books, New York.

Henry, S., Kafura, D., and Harris, K. (1981). On the relationships among three software metrics. *SIGMETRICS Perform. Eval. Rev.*, 10:81–88.

Hooimeijer, P. and Weimer, W. (2007). Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 34–43, New York, NY, USA. ACM.

Jolliffe, I. T. (2002). *Principal Component Analysis*. Springer, New York, NY, USA.

Jones, C. (1991). *Applied software measurement: assuring productivity and quality*. McGraw-Hill, Inc., New York, NY, USA.

Joshi, H., Zhang, C., Ramaswamy, S., and Bayrak, C. (2007). Local and global recency weighting approach to bug prediction. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 33, Washington, DC, USA. IEEE Computer Society.

Kataoka, Y., Imai, T., Andou, H., and Fukaya, T. (2002). A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 576–, Washington, DC, USA. IEEE Computer Society.

Kenmei, B., Antoniol, G., and di Penta, M. (2008). Trend analysis and issue prediction in large-scale open source systems. volume 0, pages 73–82, Los Alamitos, CA, USA. IEEE Computer Society.

Khoshgoftaar, T., Pandya, A., and Lanning, D. (1995). Application of neural networks for predicting program faults. *Annals of Software Engineering*, 1:141–154. 10.1007/BF02249049.

Khoshgoftaar, T. M., Allen, E. B., Goel, N., Nandi, A., and McMullan, J. (1996). Detection of software modules with high debug code churn in a very large legacy system. In *ISSRE '96: Proceedings of the The Seventh International Symposium on Software Reliability Engineering*, page 364, Washington, DC, USA. IEEE Computer Society.

Kim, S., Zimmermann, T., Whitehead Jr., E. J., and Zeller, A. (2007). Predicting faults from cached history. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 489–498, Washington, DC, USA. IEEE Computer Society.

Knab, P., Pinzger, M., and Bernstein, A. (2006). Predicting defect densities in source code files with decision tree learners. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 119–125, New York, NY, USA. ACM.

Ko, A. J. and Chilana, P. K. (2010). How Power Users Help and Hinder Open Bug Reporting. In *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*, pages 1665–1674, Atlanta, Georgia, USA. ACM.

Kohavi, R. and John, G. H. (1997). Wrappers for feature subset selection. *Artif. Intell.*, 97:273–324.

Koru, A. G. and Tian, J. J. (2005). Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products. *IEEE Trans. Softw. Eng.*, 31:625–642.

Lamkanfi, A., Demeyer, S., Giger, E., and Goethals, B. (2010). Predicting the severity of a reported bug. In *Proc. 7th IEEE Working Conf. Mining Software Repositories (MSR)*, pages 1–10.

Lessmann, S., Baesens, B., Mues, C., and Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Softw. Eng.*, 34(4):485–496.

Li, P. L., Herbsleb, J., and Shaw, M. (2005). Forecasting field defect rates using a combined time-based and metrics-based approach: A case study of openbsd. In *ISSRE '05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 193–202, Washington, DC, USA. IEEE Computer Society.

Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., and Zhai, C. (2006). Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ASID '06, pages 25–33, New York, NY, USA. ACM.

Liebchen, G. A. and Shepperd, M. (2008). Data sets and data quality in software engineering. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, PROMISE '08, pages 39–44, New York, NY, USA. ACM.

Lorenz, M. and Kidd, J. (1994). *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Lyu, M. R., editor (1996). *Handbook of software reliability engineering*. McGraw-Hill, Inc., Hightstown, NJ, USA.

McCabe, T. J. (1976). A complexity measure. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA. IEEE Computer Society Press.

Menzies, T. and Marcus, A. (2008). Automated severity assessment of software defect reports. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*.

Mockus, A. (2008). Missing data in software engineering. In Shull, F., Singer, J., and Sjberg, D. I. K., editors, *Guide to Advanced Empirical Software Engineering*, pages 185–200. Springer London. 10.1007/978-1-84800-044-5_7.

Mockus, A. and Votta, L. G. (2000a). Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintanance*, pages 120–130.

Mockus, A. and Votta, L. G. (2000b). Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, ICSM '00, pages 120–, Washington, DC, USA. IEEE Computer Society.

Moser, R., Pedrycz, W., and Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 181–190, New York, NY, USA. ACM.

Musa, J. D., Iannino, A., and Okumoto, K. (1987). *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, New York.

Nagappan, N. and Ball, T. (2005a). Static analysis tools as early indicators of pre-release defect density. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 580–586, New York, NY, USA. ACM.

Nagappan, N. and Ball, T. (2005b). Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 284–292, New York, NY, USA. ACM.

Nagappan, N., Ball, T., and Zeller, A. (2006). Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 452–461, New York, NY, USA. ACM.

Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., and Murphy, B. (2010). Change bursts as defect predictors. In *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering*.

Nelson, E. (1966). Management handbook for the estimation of computer programming costs. In *Systems Development Corporation*.

Nilsson, N. (1965). *Learning Machines*. McGraw Hill, New York, NY, USA.

Ohlsson, N. and Alberg, H. (1996). Predicting fault-prone software modules in telephone switches. *IEEE Trans. Softw. Eng.*, 22:886–894.

Opitz, D. W. (1999). Feature selection for ensembles. In *Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence*, AAAI '99/IAAI '99, pages 379–384, Menlo Park, CA, USA. American Association for Artificial Intelligence.

Ostrand, T., Weyuker, E., and Bell, R. (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355.

Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., and Wang, B. (2003). Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 465–475, Washington, DC, USA. IEEE Computer Society.

Porter, D. E. (1962). Industrial dynamics. *Science*, 135(3502):426–427.

Provost, F. and Fawcett, T. (2001). Robust classification for imprecise environments. *Machine Learning*, 42:203–231. 10.1023/A:1007601015854.

Purushothaman, R. and Perry, D. E. (2005). Toward understanding the rhetoric of small source code changes. *IEEE Trans. Softw. Eng.*, 31:511–526.

Putnam, L. H. and Myers, W. (1991). *Measures for Excellence: Reliable Software on Time, within Budget*. Prentice Hall Professional Technical Reference, 1st edition.

Quinlan, J. R. (1986). Induction of decision trees. *Mach. Learn.*, 1:81–106.

Quinlan, J. R. (1992). Learning with continuous classes. pages 343–348. World Scientific.

Quinlan, J. R. (1993). *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc.

Ratzinger, J., Sigmund, T., Vorburger, P., and Gall, H. (2007). Mining software evolution to predict refactoring. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 354–363, Washington, DC, USA. IEEE Computer Society.

Russell, S. J. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Printice Hall.

Schroter, A. (2007). Predicting defects and changes with import relations. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 31, Washington, DC, USA. IEEE Computer Society.

Schröter, A., Zimmermann, T., Premraj, R., and Zeller, A. (2006). If your bug database could talk. In *In Proceedings of the 5th International Symposium on Empirical Software Engineering, Volume II: Short Papers and Posters*, pages 18–20.

Shepperd, M. and Schofield, C. (1997). Estimating software project effort using analogies. *IEEE Trans. Softw. Eng.*, 23:736–743.

Simon, F., Steinbrückner, F., and Lewerentz, C. (2001). Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, CSMR '01, pages 30–, Washington, DC, USA. IEEE Computer Society.

Śliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA. ACM.

Song, Q., Shepperd, M., Cartwright, M., and Mair, C. (2006). Software defect association mining and defect correction effort prediction. *IEEE Trans. Softw. Eng.*, 32:69–82.

Subramanyam, R. and Krishnan, M. S. (2003). Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29:297–310.

Tai, K.-C. (1984). A program complexity metric based on data flow information in control graphs. In *Proceedings of the 7th international conference on Software engineering*, ICSE '84, pages 239–248, Piscataway, NJ, USA. IEEE Press.

Tsymbal, A. (2004). The problem of concept drift: Definitions and related work. Technical report, Department of Computer Science Trinity College.

Čubranić, D. and Murphy, G. C. (2003). Hipikat: recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 408–418, Washington, DC, USA. IEEE Computer Society.

Čubranić, D., Murphy, G. C., Singer, J., and Booth, K. S. (2005). Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31:446–465.

Vorburger, P. and Bernstein, A. (2006). Entropy-based concept shift detection. In *ICDM '06: Proceedings of the Sixth International Conference on Data Mining*, pages 1113–1118, Washington, DC, USA. IEEE Computer Society.

Wang, Y. and Witten, I. (1997). Induction of model trees for predicting continuous classes. In *Proc. of the Poster Papers of the European Conference on ML, Prague: Faculty of Informatics and Statistics, University of Economics*, pages 128–137.

Weiss, C., Premraj, R., Zimmermann, T., and Zeller, A. (2007). How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 1–, Washington, DC, USA. IEEE Computer Society.

Weston, J., Elisseeff, A., Schölkopf, B., and Tipping, M. (2003). Use of the zero norm with linear models and kernel methods. *J. Mach. Learn. Res.*, 3:1439–1461.

Widmer, G. and Kubat, M. (1993). Effective learning in dynamic environments by explicit context tracking. In *ECML '93: Proceedings of the European Conference on Machine Learning*, pages 227–243, London, UK. Springer-Verlag.

Witten, I. H. and Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann.

Zeng, H. and Rine, D. (2004). Estimation of software defects fix effort using neural networks. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - Volume 02*, COMPSAC '04, pages 20–21, Washington, DC, USA. IEEE Computer Society.

Zimmermann, T., Premraj, R., and Zeller, A. (2007). Predicting defects for eclipse. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 9, Washington, DC, USA. IEEE Computer Society.

Part IX

Appendix

Improving Defect Prediction Using Temporal Features and Non Linear Models

Abraham Bernstein
Department of Informatics
University of Zurich
Switzerland
bernstein@ifi.unizh.ch

Jayalath Ekanayake
Department of Informatics
University of Zurich
Switzerland
jayalath@ifi.unizh.ch

Martin Pinzger
Department of Informatics
University of Zurich
Switzerland
pinzger@ifi.unizh.ch

ABSTRACT

Predicting the defects in the next release of a large software system is a very valuable asset for the project manger to plan her resources. In this paper we argue that *temporal features (or aspects) of the data are central to prediction performance*. We also argue that *the use of non-linear models, as opposed to traditional regression, is necessary to uncover some of the hidden interrelationships between the features and the defects* and maintain the accuracy of the prediction in some cases.

Using data obtained from the CVS and Bugzilla repositories of the Eclipse project, we extract a number of temporal features, such as the number of revisions and number of reported issues within the last three months. We then use these data to predict both the location of defects (i.e., the classes in which defects will occur) as well as the number of reported bugs in the next month of the project. To that end we use standard tree-based induction algorithms in comparison with the traditional regression.

Our non-linear models uncover the hidden relationships between features and defects, and present them in easy to understand form. Results also show that using the temporal features our prediction model can predict whether a source file will have a defect with an accuracy of 99% (area under ROC curve 0.9251) and the number of defects with a mean absolute error of 0.019 (Spearman's correlation of 0.96).

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement; D.2.8 [**Software Engineering**]: Metrics

Keywords

Mining Software Repository, Defect Prediction, Decision Tree Learner

1. INTRODUCTION

One of the central questions in software engineering is how to write bug-free software. Given that it is virtually impossible to do so researchers are striving to develop approaches for predicting the location, number, and severity of future/hidden bugs. Such predictions can be used by software development managers to (among other things): (1) identify the most critical parts of a system that should be improved by respective restructuring, (2) try to limit the gravity of their impact by, e.g., "avoiding" the use of these parts, and/or (3) to plan testing efforts (parts with most defects should be tested most frequently).

Several approaches have been developed to predict future faults using historical data [2, 3, 5, 11], however many of them have not been evaluated or are not applicable to large software systems. Addressing these two issues our long term objective is to develop an easy-to-use tool for predicting future defects in source files akin to the Hatari tool described in [16]. In this paper we present a number of experiments to investigate the significance of temporal features and the applicability of non-linear models for predicting whether a source file will have a defect and the number of defects. A well performing prediction model is key for our tool.

For our experiments we employ six plugins of the Eclipse project. For each plugin we obtain historical data from the issue tracking system Bugzilla[1] and the version control system CVS.[2] Based on these data we compute a number of features of the actual source code, past defects (bugs), and modifications. In our experiments we test different feature sets to find out the most significant one. For the prediction of the location of defects we use a decision tree learner as has been also used in one of our previous experiments presented in [7]. For the prediction of the number of defects in source files we use a regression tree learner (in addition to traditional regression). Referring to the previous application examples we used the results of our predictions to identify the Eclipse plugins (out of the six) that should be refactored and tested with care.

The results of experiments show that the use of *temporal features* significantly improves the performance of prediction models (both, location and number of defects). Furthermore, we show how the use of *non-linear models* helps to uncover some of the non-linear relationships between features as well as between the feature and the target variables (i.e., defect location and number of defects) improving prediction performance. Our model exhibits excellent results: we are able to predict the defect location with an

[1] `http://www.bugzilla.org/`
[2] `http://www.nongnu.org/cvs/`

IWPSE'07, September 3-4, 2007, Dubrovnik, Croatia.

accuracy of 99% (given a base rate of 96.3% and 3.6%) resulting in a distribution independent area under the ROC curve of 0.9251 (see [12]). The number of defects prediction model also exhibits an excellent prediction resulting in a Spearman's correlation of 0.96 and a mean absolute error of 0.0194. Furthermore, we observed that (1) the best predictor for defects in a source file was the past existence thereof [3], (2) some features collected were actually detrimental to the overall performance, and (3) the process measures based on the change history were better predictors than the traditionally used code-metrics, which is supported by several recent studies [2, 7].

The remainder of the paper is organized as follows: After discussing the related work in the following Section 2, we describe the experimental setup in detail (Section 3), which is followed by a discussion of the results. We close with a discussion of the limitations of our study, some possible avenues of future work, and concluding remarks.

2. RELATED WORK

The historical data of software systems is a valuable asset used for research ranging from software design to software development, software maintenance, software understanding, and many more. A number of researches use the historical data of software projects for their research in the above fields. We list here few of the studies similar to our study.

Khoshgoftaar et al.[5] used a history of process metrics to predict software reliability and to prove that the number of past modifications of a source file is a significant predictor for its future faults.
Mockus et al.[9] studied a large software system to test the hypothesis that evolution data can be used to determine the changes of the software systems and to understand and predict the state of software projects. Our approach also supports this idea.
Graves et al. [2] developed statistical models to find which features of a module's change history were the best predictors for future faults. They developed a model called weighted time damp model which predicted the fault potential using changes made to the module in the past. We use similar features but employ non-linear models.
Hassan et al. [3] developed a set of heuristics which highlights the most susceptible subsystems to have a fault. The heuristics are based on the subsystems that were most frequently and most recently fixed. Our approach provides some matrices to represent the above heuristics.
Nagappan et al.[10] presented a method to predict defect density based on code churn metrics. They found out that source files with a high activity rate in the past will likely have more defects than source files with a low activity rate. They pointed out that the relative measures are better predictors for defects than the absolute measures. In our experiment, all the measures are relative and moreover we used machine learning techniques in addition to the linear regression model to predict the number of defects.
Ostrand et al. [11] used a regression model to predict the location and number of faults in large industrial software systems. The predictors for the regression model were based on the code length of the current release, and fault/modification history of the file from previous releases. Our study also supports the significance of the modification reports and the number of reported problems for defect prediction but does not support the significance of the code length.
Knab et al. [7] presented a method to predict defect densities in source code files using decision tree learners. This approach is quite smiler to our approach. However they predicted only the number of problems reported. In our models, we predict both the number of problems and the locations. They used both product and process measures for the defect prediction and revealed that process measures are more significant indicators for fault prediction than product measures, which is also supported by our findings.
Askari et al. [1] presented three probabilistic models to predict the number of defects of source files. They used an information theoretic approach and pointed out that the predictive rate of modification in a file is incremented by any modification to that file and decay exponentially. In our study we also use past modification reports as an indicator of defects.
Finally, Zimmermann et al. [18] proposed a statistical model to predict the location and the number of bugs. They used logistic regression model to predict the location of bugs and the linear regression model to predict the number of bugs. Further they heavily used product metrics such as McCabe cyclomatic complexity as predictors than process metrics. In this study, we use non-linear decision tree models to predict the location and the number of bugs and show that they are superior to linear ones. Furthermore, we heavily rely on process metrics than product metrics.

Our approach takes guidance from these approaches. It seems to be the first to combine the use of temporal features with non-linear models.

3. EXPERIMENTAL SETUP

In this section we succinctly introduce the overall experimental setup. We discuss the data used and measures used to judge the quality of the results.

3.1 The Data - CVS and Bugzilla for Eclipse

The data for the experiment was extracted from six plugins of the Eclipse open source project: `updateui`, `updatecore`, `search`, `pdeui`, `pdebuild`, and `compare`. For each plugin we considered the CVS and Bugzilla data from the first releases up to the last one released in January 2007 as provided by the MSR Mining Challenge 2007.[3] Table 1 lists the release dates and the number of files (taken from the last release).

Plugin	First Release	Last Release	#Files
updateui	Jan 03, 2001	Jan 18, 2007	757
updatecore	Jan 03, 2001	Jan 18, 2007	459
search	May 02, 2001	Jan 30, 2007	540
pdeui	Mar 26, 2001	Jan 30, 2007	1621
pdebuild	Dec 11, 2001	Jan 12, 2007	198
compare	May 02, 2001	Jan 30, 2007	315
Total			3890

Table 1: Eclipse plugins considered

Of the 3890 files we omitted 59 as they did not have a sufficient number of revisions to provide temporal information for our experiment. Other examples for exclusion

[3] `http://msr.uwaterloo.ca/msr2007/challenge/`

#	Name	Description
1	LOC	Number of lines of codes
2	LineAddedIRLAdd	Number of lines added to fix a bug relative to total number of lines added
3	LineDeletedIRLDel	Number of lines deleted to fix a bug relative to total number of line deleted
4	AlterType	Amount of modification done relative to LOC
5	AgeMonths	Age of a file in months
6	RevisionAge	Number of revisions relative to the age of a file
7	DefectReleases	Number of releases of a files with defects relative to total number of releases
8	Revision1Month	Number of revisions of a file from Dec 1 to 31 of 2006
9	DefectAppearance1Month	Number of releases of a file with defects from Dec 1 to 31 of 2006
10	ReportedI1Month	Number of reported problems of a file from Dec 1 to 31 of 2006
11	Revision2Months	Number of revisions of a file from Nov 1 to Dec 31 of 2006
12	DefectAppearance2Months	Number of releases of a file with defects from Nov 1 to Dec 31 of 2006
13	ReportedI2Months	Number of reported problems of a file from Nov 1 to Dec 31 of 2006
14	Revision3Months	Number of revisions of a file from Oct 1 to Dec 31 of 2006
15	DefectAppearance3Months	Number of releases of a file with defects from Oct 1 to Dec 31 of 2006
16	ReportedI3Month	Number of reported problems of a file from Oct 1 to Dec 31 of 2006
17	Revision5Months	Number of revisions of a file from Aug 1 to Dec 31 of 2006
18	DefectAppearance5Months	Number of releases of a file with defects from Aug 1 to Dec 31 of 2006
19	ReportedI5Month	Number of reported problems from Aug 1 to Dec 31 of 2006
20	ReportedIssues	Total number of reported problems
21	Releases	Total number of releases
22	RevisionAuthor	Number of revisions per author

Table 2: The features (or measures) used in our experiment

were files with modification reports that do not contain lines added/deleted information or with a wrong or unavailable release date. We exported all the information into the evolution ontology format EvoOnt data [6], which integrates the code, release, and bug information in a single knowledge base. For each of the investigated 3831 source files we used the information in the EvoOnt knowledge base to compute the number of lines of code (LOC) code and several process features (or measures) as listed in Table 2. Features 8–19 contain temporal/historical information about the project. Essentially, they consider different sizes of windows (1, 2, 3, and 5 months) backwards from the December 2006 releases. If a defect is not fixed in one release and transferred to later releases, then we count them in all releases where they occur.

Since lines of codes are added/deleted both when fixing a bug and when adding new features they need to be separated. Features LineAddedIRLAdd and LineDeletedIRLDel represent the number of lines added/deleted to fix a bug relative to total number of lines added/deleted.

Feature 4, AlterType, classifies each modification into large, medium, and small according its size relative to the lines of code modified in the source files. If the sum of lines added and deleted is more than double of the code length then AlterType of this modification is large. If the modification relative to the code length is between 1 and 2 then Altertype is medium. If the size of the change is below 1 than AlterType is small. This reflects the way modifications are handled by CVS, which stores for a modified line 1 line deleted and 1 line added.

3.2 Experimental Procedure

All experiments were carried out using the Weka data mining toolkit [17]. To test the quality for our prediction models we computed the features shown in Table 2 once for releases until December 31 2006 for learning/inducing the model – the *training set* – and once for the period until the end of January as a *test set*.

Since choosing a good feature set for the prediction model is imperative for a good prediction performance we used a number of *wrapper-based feature selection* methods such as sequential forward selection [8]. These methods compare the prediction performance of different subsets of the features within the training set to find the best performing subset.

The best performing subset of features was then used to induce the prediction model, which was then tested on the test set. This procedure ensures that only information available on December 31, 2006 was used to predict the location of defects or the number of bugs in January of 2007.

3.3 Performance Measures

For the location prediction experiment we learned a class probability estimation model (CPE), which computes the probability distribution over the two possible classes: hasBug and hasNoBug. Since CPE's are usually used to predict classes we picked the class with the higher probability and computed the confusion matrix of the model, which can (partially) be summarized with accuracy of the model's classification. The problem of the accuracy as a measure is that it does not relate the prediction the prior probability of the classes. This is especially problematic in heavily skewed distributions such as the one we have. Therefore, we also used the receiver operating characteristics (ROC) and the area under the ROC curve, which relate the true-positive rate to the false-positive rate resulting in a measure uninfluenced of the prior (or distribution) [12, 17].

Given the skewed distribution the traditional Pearson correlation is inappropriate. For the regression experiment we, therefore, report Spearman's Rank correlation (ρ), root mean squared error (RMSE), and mean absolute error (MAE).

4. EXPERIMENTS

In our experiments we investigate the suitability of our approach for two tasks. First, we looked if the features selected are sufficient to predict the files that will have *defects in future versions*. Second, we explore how well our approach predicts the *number of bugs* per each file.

4.1 Defect Location Prediction

The goal of this experiment is to predict the locations of defects of source code files. To that end we learn a model using the training data that predicts the probability of defect occurrence for any given file from the test set. We used Weka's J48 decision tree learner (a re-implementation of C4.5 [13]). To test our proposition—that temporal features would improve the prediction quality—we learned the model with different base-sets of features either using no temporal data whatsoever (i.e., excluding features 8-19 of Table 2) or using the temporal features for different window sizes of 1, 2, 3, and 5 months (i.e., choosing a selection of features 8-19 representing the window size under investigation).

Table 3 summarizes the results of these experiments. It shows the list of features chosen by the feature selection method, the accuracy, and the area under the ROC curve of the prediction for each of the learned models. It is interesting to observe that the only feature chosen for all models is the `LineAddedIRLAdd` (Feature 2), which relates the numbers of lines added due to bug fixing to the number of lines added due to adding new features. Even though this feature is chosen by all models it does not seem to play a pivotal role in the models, as it does not show in none of the trees' first two levels. Another interesting observation is the dominance of the temporal features (numbers 8-19): not only do they get chosen whenever possible, they also show up at the root of the tree (see column 3) whenever available.

When looking at the target performance measures accuracy and area under the ROC curve (AUC) we clearly see the dominance of the prediction that can take advantage of temporal features compared to the one that cannot. In terms of accuracy we can clearly see that temporal information boosts the performance, but that more recent temporal data is more useful than older one. We can, hence, hypothesize that modules with bugs are likely to have bugs in later versions, but that over longer periods of time those bugs could be fixed. In other words: Bugs are likely to survive some versions, but are fixed after some.

One might argue that the difference between 96.58% (no temporal features) and 99.16% (significant features) in accuracy is not significant enough to warrant the computation of the temporal measures. Note, however, that the sole use of accuracies is misleading since they are heavily dependent on the prior distribution of the data. In our case, where the class distribution is highly skewed (we have 140 buggy classes versus 3691 non-buggy ones), it is simple to attain a high accuracy: "just" assigning "non-buggy" to every file (the default strategy) one gets an accuracy of 96.35% ($= \frac{3691}{3691+140}$) according to the confusion matrix for the best model (including significant features) shown in Table 4. Hence, the use of accuracy as a measure for the quality of the prediction is misleading. We, therefore, computed the receiver operating characteristics (ROC) for each of the methods and the area under the ROC-curve (AUC), which both provide a prior-independent approach for comparing the quality of a predictor [12].

Figure 1 graphs the ROC curves for all the chosen methods. The x-axis shows the false-positive rate and the y-axis the true positive rate. Note, that a random bug assignment is also shown as a line form the origin $(0, 0)$ to $(1, 1)$ and that the ideal ROC curve would be going from the origin straight up to $(0, 1)$ and then to $(1, 1)$. The Figure clearly shows that all prediction methods provide a significant lift in predictive quality over the random assignment. But the methods have very interesting differences in terms of quality. Since one method dominates another when its ROC-curve is closer towards the upper left corner, we can see how the non-temporal prediction model is dominated along almost the whole frontier by the temporal models. The figure also shows how the method using significant features dominates the other methods along almost the whole frontier whilst employing fewer features (see Table 3). Lastly note, that the dominance of the ROC-curve is reflected by a larger area under the ROC curve (AUC) as listed in Table 3.

	predicted buggy	predicted bug free
has bugs	117	23
has no bugs	9	3682

Table 4: Confusion Matrix for the significant features model

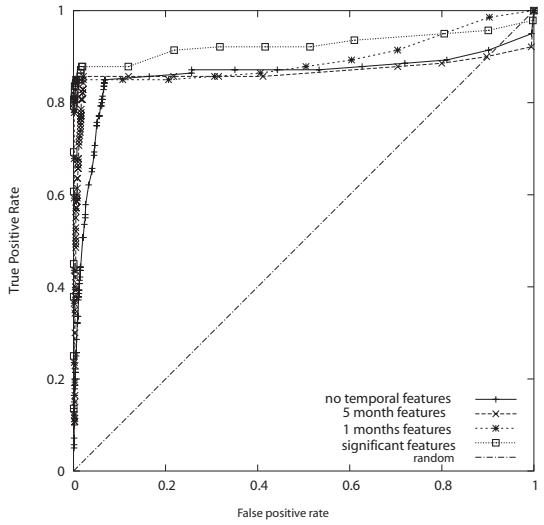


Figure 1: ROC-curves of defect prediction methods.

To further improve our understanding of the structure of the prediction methods, we succinctly compare the two top levels of the prediction trees for the non-temporal, the 1-month, and the significant feature model. As the top levels of the trees depicted in Figure 2 show even the model without temporal features (a) heavily relies on the quasi temporal feature `DefectReleases`, which computes the fraction of past releases with bugs. The next most important non-temporal feature seems to be `LineDeletedIRLDel` signifying the importance to distinguish between changes due to bug fixing versus changes due to the addition of new features. Not that this seems to be a very important distinction, as the related `LineAddedIRLAdd` feature is the most important non-temporal feature in the tree (b).

Name of method	Features chosen	Root Node of Tree	Accuracy	Area under ROC curve
no temporal features	2,3,4,6,7,20,21,22	`DefectReleases`	96.5805%	0.8611
1-month features	2,8,10,20,22	`ReportedI1Month`	99.1125%	0.8948
2-months features	2,4,11,12,13,20,22	`ReportedI2Month`	98.8776 %	0.8933
3–months features	2,3,4,6,7,14,15,16,20,21,22	`DefectAppearance3Months`	98.6427%	0.9039
5-months features	2,3,4,5,6,7,17,19,20,21,22	`ReportedI5Months`	97.7813%	0.8663
significant features	2,3,8,9,11,16,19	`DefectAppearance1Month`	99.1647%	0.9251

Table 3: Results of different models for defect location prediction (Accuracy of default strategy 96.35%)

```
DefectReleases <= 5.263158: NO (3423.0/11.0)
DefectReleases > 5.263158
|   DefectReleases <= 21.95122
|   |   LineDeletedIRLDEl <= 17.518248: NO (222.67/35.0)
|   |   LineDeletedIRLDEl > 17.518248
|   |   |   Releases <= 206: NO (34.33/10.0)
|   |   |   Releases > 206
|   |   |   |   RevisionAuthor <= 3.125: YES (7.0)
|   |   |   |   RevisionAuthor > 3.125: NO (3.0/1.0)
|   DefectReleases > 21.95122
|   |   DefectReleases <= 67.142857
|   |   |   AlterType = large
```

(a) No temporal features

```
ReportedIssueslMonth <= 0: NO (3692.0/21.0)
ReportedIssuesl Month > 0
|   RevisionlMonth <= 1: YES (105.0/2.0)
|   RevisionlMonth > 1
|   |   LineAddedIRLADD <= 3.636364: NO (16. 0/2.0)
|   |   LineAddedIRLADD > 3.636364
|   |   |   ReportedIssues <= 7: YES (15.0/1.0)
|   |   |   ReportedIssues > 7: NO (3.0)
```

(b) 1-month temporal features

```
DefectAppearancelMonth <= 0
|   ReportedI5months <= 0: NO (3599.0/9.0)
|   ReportedI5months > 0
|   |   Revision2Months <= 4: NO (86.0/7.0)
|   |   Revision2Months > 4
|   |   |   LineAddedIRLADD <= 1.359223: NO (2.0)
|   |   |   LineAddedIRLADD > 1.359223: YES (5.0)
DefectAppearancelMonth > 0
|   RevisionlMonth <= 1: YES (105.0/2.0)
|   RevisionlMonth > 1
```

(c) significant features

Figure 2: Top levels of induced defect location trees

Summarizing, we can say that the experiment for defect location prediction clearly shows that one *can, indeed, predict the location of bugs with a high accuracy.* We can also say that this accuracy bases (to a large extent) on temporal features. We hypothesize that one reason for the effectiveness of temporal values is that bugs usually survive more than one release. Other reasons might be the fact that complicated/complex or badly engineered classes are likely to exhibit bugs repeatedly unless they are re-engineered. Furthermore, we observe that the most important non-temporal features for prediction are the relation between line changes due to feature additions versus line changes due to bug fixing in the past – a type of feature not yet largely investigated in the literature, which clearly deserves more attention.

4.2 Predicting the Number of Bugs

The goal of the second group of experiments is to establish if our approach can amply predict the number of bugs that occur in any given file. This task is more difficult than the last, as it not only has to predict the existence of bugs (i.e., if $\#bugs > 0$) but the actual numbers of bugs. Since we believe that the task of predicting the number of bugs exhibits non-linear properties (a belief, for which we show evidence in section 4.3) we decided to use a non-linear regression approach. To preserve the comprehensibility of the model as well as the comparability of the model to the defect location prediction above we chose the Weka implementation of the M5 tree regression algorithm [14] called M5P. A regression tree model combines a decision tree an a linear regression by partitioning the feature space with a decision tree and then providing a linear regression equation for each of the tree's leafs. The model can, thus, predict a number by assigning any instance (i.e., entity to predict) to a leaf and then performing the associated regression to compute a number. This approach has the advantage that the regressions at the leafs do not have to be linearly connected – the tree provides the non-linear partition, the linear regressions predict the number.

The predictive power of temporal features. To test our proposition – that temporal features improve the prediction quality – we followed the same procedure as above: we learned the model with different base-sets of features either using no temporal data whatsoever (i.e., excluding features 8-19 of Table 2) or using the temporal features for different window sizes of 1, 2, 3, and 5 months.
Table 5 summarizes the results for this comparison. Like Table 3 it Lists the name of the model, the features chosen by the feature selection algorithm and the root node of the regression tree. As performance measures it lists the Pearson correlation between the prediction and the actual data, the mean absolute error (MEA), and the root mean square error (RMSE). The results mostly mirror the ones form the location prediction experiments.
The models that can rely on the temporal features do so and even use it as the main feature for the decision tree. In contrast to the the location prediction, though, the root nodes of the tree do not have the most recent available number of reported issues or bugs (i.e., `ReportedI1Month`, `ReportedI2Month`, `ReportedI2Month`, `DefectAppearance1-Months`, or alternatively `DefectAppearance3Months`), but exclusively uses the number of available (i.e., `RevisionXMonth`, where `X` is the most recent available number for learning). While this is surprising at the surface further investigation clarifies the issue: when investigating the features chosen by the feature selection algorithm we can clearly see that the elements chosen as root nodes in the defect location prediction are used in the defect number prediction. In contrast to the defect location prediction they are not at the root of the partitioning decision tree but are mostly used in the regression function at the leafs. Consider, for example, the model induced for significant-features model as shown in Figure 3. At the top we can clearly see the decision tree that partitions the feature space using only some of the features. Below, the figure shows the first of 8 linear regression models. this par-

Name of model	Features chosen	Root Node	Spearman's ρ	MAE	RMSE
no temporal features	3,5,7,20,21,22	`LineDeletedIRLDel`	0.863	0.0524	0.1898
1-months features	2,3,5,6,7,10,8,9,10,20,21,22	`Revision1Month`	0.941	0.0226	0.1272
2-months features	3,5,6,7,11,12,13,21,22	`Revision2Months`	0.950	0.0249	0.133
3-months features	2,5,7,14,15,16,21	`Revision3Months`	0.966	0.0241	0.1298
5-months features	2,3,5,17,18,19,22	`Revision5Months`	0.942	0.0326	0.1575
significant features	5,7,8,9,12,14,15,16,21,22	`Revision1Month`	0.963	0.0194	0.1119

Table 5: Results of different models for defect location prediction with M5P

ticular model is called if the rule at the root of the tree (`Revision1Month` $\leqslant 0.5$) is true. As the regression shows it uses the root node of the defect prediction decision tree `DefectAppearance1Month` with the second strongest weight in the regression.

```
Revision1Month <= 0.5 : LM1 (1348/0%)
Revision1Month >  0.5 :
|   LineAddedIRLADD <= 0.098 :
|   |   AgeMonths <= 33.667 :
|   |   |   Releases <= 65.5 : LM2 (343/0%)
|   |   |   Releases >  65.5 :
|   |   |   |   AgeMonths <= 15.95 : LM3 (112/87.619%)
|   |   |   |   AgeMonths >  15.95 : LM4 (266/26.955%)
|   |   AgeMonths >  33.667 : LM5 (975/0%)
|   LineAddedIRLADD >  0.098 :
|   |   Defectappearance3Months <= 0.5 : LM6 (619/42.644%)
|   |   Defectappearance3Months >  0.5 :
|   |   |   Revison3Months <= 1.5 : LM7 (81/171.567%)
|   |   |   Revison3Months >  1.5 : LM8 (87/210.532%)

LM num: 1
NumberofErroresLastMonth =
        0 * LineAddedIRLADD
        + 0 * AgeMonths
        + 0.0005 * Revison3Months
        - 0.0005 * Defectappearance3Months
        - 0.0013 * ReportedI3Months
        - 0 * Releases
        + 0 * RevisionAuthor
        - 0.0002 * Revision5Months
        + 0.0002 * DefectAppearance5Months
        - 0.0002 * Revision1Month
        + 0.0019 * DefectAppearance1Month
        + 0.0043 * ReportedI2Months
        - 0.0003
```

Figure 3: Excerpt of bug prediction model relying on significant features

Like in the bug prediction case Table 5 also clearly shows how the models with temporal features dominate the model without them. The difference in the Spearman's ρ (0.963 for temporal features vs. 0.863 without temporal features) is striking. The error rates MAE and RMSE mirror this behavior. Therefore, the results support our argument that the temporal data improve the accuracy of prediction model.

Exploring the prediction error. A closer look at the error rates in Table 5 also reveals that the RMSE is an order of magnitude larger than the MAE for all the models. This indicates that there are some large errors, which weigh in more heavily in the RMSE. Table 6 shows the histogram analysis of residual error of the significant-features model. As the table shows the bulk of the prediction has no (74.07%) or little (i.e., error $\leqslant 0.5$; in 98.69%). Nonetheless, a few predictions exhibit an error larger than 1. It is these predictions that mostly influence the error. When removing the file with a prediction error of 2.93, the MAE is lowered to 0.0194, but the RMSE is lowered to 0.0014, a full order of magnitude smaller. It is, hence, this one outlier that mostly contributes to the RSME. When, furthermore, removing all 5 files with an error larger than 1 we get a MAE of 0.0177 and a RMSE of 0.0095. We can, thus, conclude that the prediction error of our method is, in general, very small.

Error Interval	Frequency	Absolute	Cumulative
0	2838	74.08 %	74.08 %
$0 < e \leqslant 0.5$	943	24.61 %	98.69 %
$0.5 < e \leqslant 1$	45	1.17 %	99.87 %
$1 < e \leqslant 1.5$	4	0.10 %	99.97 %
$1.5 < e \leqslant 2$	0	0 %	99.97 %
$2 < e \leqslant 2.5$	0	0 %	99.97 %
$2.5 < e \leqslant 3$	1	0.03 %	100.00 %

Table 6: Residual error histogram for significant-feature model

Comparison with other defect predictions using the same data set The MSR Mining Challenge 2007,[4] which provided the data we used for our study, had a similar task as its Challenge #2. The main difference between our approach and the challenge task is that we chose to make our predictions on the file level and the Challenge task required participants to predict the number of bugs for 32 plug-ins[5] (i.e., summarizing the bugs for all their classes). Two methods, one in two versions were submitted to the mining challenge. `C-ESSEN` by Adrian Schröter [15] predicted the bugs based on the import statements used in the files. This is a measure we did not use at all. ULAR by Joshi et. al [4] uses features computed in the last month to make a prediction for next month. A second version of ULAR extends those predictions with a trend analysis. Last, an ad-hoc method used as an comparison by Thomas Zimmermann (called 1 Year ago) simply takes the measures from 2006 to make the prediction for 2007. Table 7 shows the Spearman's rank correlation (ρ) for all the methods as well as our significant-feature model. The results clearly show that our approach is better at ranking the files according to their expected bugs. The ranking, rather than the precise prediction of the number of bugs, is actually an important task when one tries to make an optimal assignment of resources (i.e., programmers) to tasks (i.e., the fixing of bugs) [12]. Note, however, that the other models are making their prediction on 32 modules whereas we limit ourselves to only 6, which is a much simpler task.

Summarizing, we can say that our non-linear bug prediction

[4] `http://msr.uwaterloo.ca/msr2007/challenge/`

[5] `http://msr.uwaterloo.ca/msr2007/challenge/plugins.txt`

Model	n	ρ
C-ESSEN (imports) [15]	32	0.67
ULAR (Last month + trends) [4]	32	0.81
ULAR (Last months) [4]	32	0.84
1 Year Ago	32	0.91
significant-features model	6	1.00

Table 7: Spearman's ρ for MSR Mining Challenge 2007 results, where n is the number of components

model supports our proposition that temporal features are imperative for an accurate prediction – without them the Spearman's rank correlation ρ between the predictions and the actual error numbers is lowered from 0.963 to 0.863. Second, we can clearly see how our model is highly accurate for most predictions and that most of the residual error is introduced by 5 predictions of 3831. Third, we tried to compare the performance of our approach to similar tasks (as we did not find any work on the same task): We find that our approach exhibits a superior performance compared to others with respect to the Spearman's rank correlation.

4.3 The Predictive power of linear and non-linear prediction methods

The second of our guiding propositions is that non-linear models should provide a superior prediction that the usually used linear ones. Specifically, we stated that the non-linear models are able to exploit the non-linear relationships between the features to make more accurate bug number predictions. By non-linear we mean here a relationship that cannot be captured by a weighted sum of simple, continuous functions of the single features (as done by a linear regression), but may require functions of two or more features. To explore this hypothesis we re-ran experiments outlined in sub-section 4.2 with a standard linear regression algorithm.

Table 8 shows the results of this analysis comparing the Spearman's rank correlation (ρ), the mean absolute error (MAE), and the root mean squared error (RSME) for the linear model (LM) – a standard linear regression – and the non-linear model in the form of the M5P algorithm. The results show that the non-linear significantly outperforms the linear model for all performance measures (results for pairwise t-test significant at: $p = 1.09\%$ for ρ, $p = 0.29\%$ for MEA, and $p = 2.42\%$ for RMSE). The dominance is, however, not constant. For the data sets without temporal features the LM and M5P have a very similar performance. The more recent temporal features the more pronounced is the dominance of the non-linear model. This would lead us to hypothesize that the non-temporal features exhibit a non-linear relationship to the number of bugs. If we explore the actual model this hypothesis is confirmed. Consider again Figure 3, which shows the bug prediction model for significant-features. As the model clearly shows the temporal features are heavily used within the non-linear element of the model: the decision tree that partitions the feature space. Nonetheless, the temporal features are also reused in the linear part of the model: the leaf-based regressions. We can, thus, conclude that (1) the temporal features have both linear and non-linear elements with respect to the number of bugs and (2) the M5P's capability to exploit both linear and non-linear elements clearly results in more accurate results.

4.4 Identifying the critical Eclipse plugins

We applied the best performing prediction model to identify the most critical Eclipse plugins (out of the six). These plugins need be considered first when planning refactoring and testing efforts. With critical we mean plugins for which our model predicts the highest number of bugs for January 2007. Table 9 lists the results with the actual number of bugs, the predicted number of bugs, and the accuracy of the prediction model.

Pugin	#Actual	#Predicted	Accuracy
pdeui	83	68.8999	83.0119%
compare	36	29.5561	82.1002%
pdebuild	20	16.7421	83.7106%
updateui	10	8.6371	86.3718%
updatecore	8	7.1928	89.9104%
search	1	1.0663	93.7836%

Table 9: Predicted and actual number of bugs for the six Eclipse plugins in January 2007.

From a managers point of view the number of predicted bugs clearly indicates that refactoring as well as testing effort needs to be dedicated to the two plugins `pdeui` and `compare`. In particular, `pdeui` is indicated as a critical plugin that according to our model will be affected by around 69 bugs in January 2007. This mirrors the actual number of bugs, which was 83. From that we conclude that such predictions provide a valuable input for software project managers to plan refactorings and tests.

5. LIMITATIONS AND CONCLUSIONS

Our findings for the Eclipse evolution data are very promising. The use of a non-linear model basing on temporal features selected by an automated feature selection algorithm could predict defect location and numbers with a very high accuracy. The findings are, however, hampered with a number of limitations.

First and foremost, the chosen *Eclipse data set represents only one project-family.* While we followed good data mining practice to ensure the generalizability of the our findings the data might behave Eclipse idiosyncratic such as a common culture of bug-reporting or code documentation/fixing, programming language dependencies (Eclipse only uses Java), etc. Furthermore, we only looked at the predictions for the last month (January 2007) of the data set. We intend ascertain the generalizability of our findings by (1) exploring the quality of the prediction for other months within the Eclipse data set and for other projects altogether.

Second, we *"only" used one off-the-shelf feature selection and non-linear induction algorithm.* It might, therefore, be that the resulting feature set and model are suboptimal. Following good data analysis practice we should try a whole set of algorithms to determine the most predictive model – a task that we will undertake in the near future. Nonetheless, we are confident that the use of other algorithms will not substantially change our findings. Much more we expect them to potentially make them even more pronounced than currently.

Third, our candidate features were chosen by our study of

Name of model	LM ρ	LM MAE	LM RMSE	M5P ρ	M5P MAE	M5P RMSE
without temporal features	0.844	0.0569	0.1902	0.863	0.0524	0.1898
1-Month	0.935	0.0306	0.1311	0.941	0.0226	0.1272
2-Months	0.919	0.039	0.1421	0.950	0.0249	0.133
3-Months	0.891	0.0471	0.1523	0.966	0.0241	0.1298
5-Months	0.918	0.0423	0.1611	0.942	0.0326	0.1575
Significant Features	0.929	0.0319	0.1227	0.963	0.0194	0.1119

Table 8: Comparison of linear model (LM) and Non-linear model (M5P), ρ is the Spearman's rank corr.

the literature and some of our own thoughts regarding temporal features. In order to ensure an optimal performance of the resulting models we need to *explore the full space of possibly applicable measures* (or features) reported in the literature. We hope to investigate the full feature space in the future. Like with the feature selection, however, we think that such an exploration would make our finding more pronounced but not change the inferred conclusions.

Last and most importantly, our attempt could be seen as a post-prediction rather than a pure prediction. After all, we could employ some "current" information in building our models. We intend to address this problem in the future by completely temporally disentangling training from test set. In the future we intend to *embed this approach into a tool, which seamlessly integrates into an IDE* and highlights files that have a high probability of defects or a large number of bugs. Such an integration would simplify the use of the algorithm by software managers and developers, which would allow to investigate their use in practice.

We also intend to pursue the avenue of temporal dependencies/relationship between code/bug-measures and future performance. To that end we also intend to explore the use of temporal data mining techniques such a Markov models. In closing we should highlight that *our approach - employing temporal features and non-linear models for defect prediction shows a clear advantage over others*. We hope that this method will help to contribute to improved bug number predictions and, therefore, help to ensure the development of software with fewer bugs.

6. ACKNOWLEDGMENTS

This work was partially supported by a grant of the Sri Lankan government. We would like to thank Thomas Zimmermann for making available the data for the 2007 MSR mining challenge as well as the anonymous reviewers whose comments helped to improve this paper.

7. REFERENCES

[1] M. Askari and R. Holt. Information theoretic evaluation of change prediction models for large-scale software. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 126–132, New York, NY, USA, 2006. ACM Press.

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

[3] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 263–272, Washington, DC, USA, 2005. IEEE Computer Society.

[4] H. Joshi, C. Zhang, S. Ramaswamy, and C. Bayrak. Local and global recency weighting approach to bug prediction. In *MSR 2007: International Workshop on Mining Software Repositories*, 2007.

[5] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, pages 364–371, White Plains, NY, 1996. IEEECS.

[6] C. Kiefer, A. Bernstein, and J. Tappolet. Analyzing software with isparql. In *Proceedings of the 3rd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2007)*. Springer, June 2007. to appear.

[7] P. Knab, M. Pinzger, and A. Bernstein. Predicting defect densities in source code files with decision tree learners. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 119–125, New York, NY, USA, 2006. ACM Press.

[8] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.

[9] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 120, Washington, DC, USA, 2000. IEEE Computer Society.

[10] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *ICSE '05: Proceedings of the 27th international conference on Software engineering, p580–586*, 2005.

[11] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, 2005.

[12] F. J. Provost and T. Fawcett. Robust classification for imprecise environments. volume 42, pages 203–231, 2001.

[13] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

[14] R. J. Quinlan. Learning with continuous classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, Singapore, 1992.

[15] A. Schröter. Predicting defects and changes with import relations. In *Proceedings of MSR 2007: International Workshop on Mining Software Repositories*, 2007.

[16] J. Sliwerski, T. Zimmermann, and A. Zeller. Hatari: Raising risk awareness (research demonstration). In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 107–110. ACM, September 2005.

[17] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, second edition, 2005.

[18] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse, May 2007.

Tracking Concept Drift of Software Projects Using Defect Prediction Quality

Jayalath Ekanayake*, Jonas Tappolet*, Harald C. Gall+, Abraham Bernstein*
*Dynamic and Distributed Systems Group, +Software Evolution and Architecture Lab
Department of Informatics, University of Zurich
{jayalath, tappolet, gall, bernstein}@ifi.uzh.ch

Abstract

Defect prediction is an important task in the mining of software repositories, but the quality of predictions varies strongly within and across software projects. In this paper we investigate the reasons why the prediction quality is so fluctuating due to the altering nature of the bug (or defect) fixing process. Therefore, we adopt the notion of a concept drift*, which denotes that the defect prediction model has become unsuitable as set of influencing features has changed – usually due to a change in the underlying bug generation process (i.e., the concept). We explore four open source projects (Eclipse, OpenOffice, Netbeans and Mozilla) and construct file-level and project-level features for each of them from their respective* CVS *and* Bugzilla *repositories. We then use this data to build defect prediction models and visualize the prediction quality along the time axis. These visualizations allow us to identify concept drifts and – as a consequence – phases of stability and instability expressed in the level of defect prediction quality. Further, we identify those project features, which are influencing the defect prediction quality using both a tree induction-algorithm and a linear regression model. Our experiments uncover that software systems are subject to considerable concept drifts in their evolution history. Specifically, we observe that the change in number of authors editing a file and the number of defects fixed by them contribute to a project's concept drift and therefore influence the defect prediction quality. Our findings suggest that project managers using defect prediction models for decision making should be aware of the actual phase of stability or instability due to a potential concept drift.*

1. Introduction

In mining software repositories, many different approaches have been developed to predict the number and location of future bugs in source code (*e.g.*, [1], [2], [3], [4], [5]). Such predictions can help a project manager to quantitatively plan and steer the project according to the expected number of bugs and their bug-fixing effort. But bug prediction can also be helpful in a qualitative way whenever the defect location is predicted: testing efforts can then be accomplished with a focus on the predicted bug locations. All of the above mentioned approaches use the history data of a software project to predict defects in the next release. Features (or variables) are extracted from the raw data. These features (from the learning period) are then used together with the goal values (*i.e.*, bug or no bug) to learn a prediction model. To evaluate such a model, it is fed with data from another time period and the predicted values are compared with the observed ones facilitating an accuracy measure.

The common downside of these approaches is their temporally coarse evaluations. Usually, a bug prediction algorithm is evaluated, in terms of accuracy, in only one or several different points in time. Such selective (insular) analyses make generalizations of the prediction methods difficult: it postulates that the evolution of a project and its data is more or less stable over time.

In our approach we hypothesize that a project passes several alternating phases of stability and instability. Instability can be seen as a sudden change of influencing factors. These factors can be of various kind such as a changing number of developers, the use of a new development tool or even political or economical events (financial crisis, presidential elections) *etc*.

As a consequence, the concept (*i.e.*, the bug generation process) we are trying to learn changes, resulting in a phenomenon called *concept drift* [6]. Obviously, concept drifts can invalidate a learned bug prediction model and lead to less accurate predictions as time progresses. Our goal is to identify and locate concept drifts that affect the accuracy of defect prediction algorithms. For that reason, *our measure for stability and instability of the concept is the quality of the defect prediction*. In a stable phase, the history data is a good predictor for future bugs; analogously, in an unstable phase, the prediction quality will significantly decrease and become unreliable for effort and resource allocation.

Our approach can be summarized as follows. To uncover stable and unstable phases we apply our bug prediction algorithm continuously over time to the data. We provide the algorithm with different temporally sampled feature sets that reflect a changing length of history data available. Hence, for each possible prediction time we evaluate the

Partial support for Jayalath Ekanayake provided by IRQUE fund of the Sabaragamuwa University of Sri Lanka
Partial support for Jonas Tappolet provided by Swiss National Science Foundation award number 200021-112330

quality of bug prediction models learned from every possible (consecutive) period in its past. For example, in month 35 of a given project we use data from the past 34 months to conduct 34 different bug prediction runs each leading its own accuracy value. This method allows us to visualize concept drifts and show that there are indeed phases in a project where a bug prediction is almost useless (with respect to accuracy) and, hence, a project manager should not rely on it. Furthermore, this approach allows us to identify the influencing features that have the potential to serve as early indicators for upcoming concept drifts.

The remainder of the paper is organized as follows: After discussing some related work in Section 2, we describe the experimental setup in Section 3, followed by a discussion of experiments and results. We close with limitations of our study, some possible avenues of future work, and concluding remarks.

2. Related Work

A number of researchers have used the historical data of software projects for different kinds of prediction models. To the best of our knowledge, there is no prior work investigating possible *concept drift* in software projects. However, we here discuss several studies about defect prediction and, as well, related work exploring *concept drift* in different domains.

Bernstein *et al*. [1] used Eclipse's history of product metrics to predict defects. However, the learned model is not evaluated in a temporally continuous way. Instead, only a couple of discontiguous points in time are considered. Since this is our previous work, we use a similar approach to predict the defects in the current experiments as well.

Khoshgoftaar *et al*. [3] used a history of process metrics to predict software reliability and to prove that the number of past modifications of a source file is a significant predictor for its future faults. We also use similar set of features for our work.

Mockus *et al*. [7] studied a large software system to test the hypothesis that evolution data can be used to determine the changes of the software systems and to understand and predict the state of software projects.

Graves *et al*. [2] developed statistical models to determine which features of a module's change history were the best predictors for future faults. They developed a model called *weighted time damp model* which predicted the fault potential by using changes made to the module in the past. We use similar features but we predict the location of the defect.

Hassan *et al*. [4] developed a set of heuristics which highlights the most susceptible subsystems to have a fault. The heuristics are based on the subsystems that were most frequently and most recently fixed. We also compute some of the features that represent the above heuristics for our models. We see most of these features frequently used by our prediction models in stable periods of the projects but not in instable periods.

Nagappan *et al*. [8] presented a method to predict the defect density based on code churn metrics. They concluded that source files with a high activity rate in the past will likely have more defects than source files with a low activity rate. We also added this particular feature set to our prediction models. But none of these features seemed to be of significant influence to a possible explanation of *concept drift*.

Ostrand *et al*. [5] used a regression model to predict the location and number of faults in large industrial software systems. Their predictors for the regression model were based on the code length of the current release and the fault / modification history of the file from previous releases. Although we don't use source code metrics in our study, we extensively use fault / modification histories.

Knab *et al*. [9] predicted defect densities in source code files using decision tree learners. This approach is quite similar to our approach. However, they predicted the number of problems reported. In our models, we predict defect locations. They used both product and process metrics and revealed that process metrics are more significant than product metrics for fault predictions. The model and features, with the exception of the product features, are quite similar to our work. However, they evaluated the model only in very few points in time.

Zimmermann *et al*. [10] proposed a statistical model to predict the location and the number of bugs. They used a logistic regression model to predict the location of bugs and a linear regression model to predict the number of bugs. Further they heavily used product metrics such as *McCabe's Cyclomatic Complexity* as predictors rather than process metrics. In our study, we use decision tree models to predict the location of bugs. Furthermore, we fully rely on process metrics.

Kim *et al*. [11] assumed that faults do not occur in isolation, but rather in bursts of several related faults. They basically considered any location recently changed or recently added together with the known bug is likely to be buggy. We also use some similar metrics such as *chanceBug* in our prediction models.

Brooks *et al*. [12] described in their famous book that adding people to a late project makes it even later. Even our study shows that the number of authors is influencing the stability of the projects.

Tsymbal [6] provided a survey on *concept drift* research in many domains. He argued that in the real world concepts are often not stable but changing over time. He showed typical examples such as weather prediction rules and customer preferences. Furthermore, he mentioned that underlying data distribution may change as well. Also he observed the models built on old data to be inconsistent with new data

and, therefore, regular updating of these models is necessary.

Harries *et al*. [13] explored concept drift in financial time series by using machine learning algorithms. We use a similar approach but with software history data to identify the concept drift.

Widmer *et al*. [14] uncovered from daily experience that the meaning of many concepts heavily depend on implicit context. Changes in that context can cause radical changes in the concept. We argue that the same effect can be observed in software systems.

Kenmei *et al*. [15] showed that the further you go in time the worst will be the prediction, which is also supported by our results.

As a closing remark for this section we like to point out that the idea of *concept drift* per se is not new to the research community. However, software projects have never been subject to such analyses, which is a gap we try to close in this work.

3. Experimental Setup

In this section we succinctly introduce the overall experimental setup. We present the data used, its acquisition method, and the measures used to evaluate the quality of the results.

3.1. The Data: CVS and Bugzilla for Eclipse, Netbeans, Mozilla, and Open Office

The data for the experiments was extracted from the four open source software projects Eclipse, Netbeans, Mozilla and Open Office. We collected the information provided by CVS and Bugzilla systems for each of the projects. The reason behind selecting these four projects is their long development history (>6 years) that is essential for this kind of analysis to ensure the gathering of multiple developments cycles and their possibly associated drifts. For classification, we use only issues which are marked as defects in the bug database. We understand authorship in terms of the person who brought the changed code into the versioning system rather than the developer who actually wrote the code. This is a necessary simplification since we do not consider the content of files which would shed some light on the real authorship. Table 1 shows an overview of the observation periods and the number of files considered in this work.

Project	First Release	Last Release	#Files
Eclipse	2001-01-31	2007-06-30	9948
Mozilla	2001-01-31	2008-02-29	1896
Netbeans	2001-01-31	2007-06-30	38301
Open Office	2001-01-31	2008-04-30	1847
Total			51,992

Table 1. Considered data sources and time spans.

For all files we exported the history information within the investigated time frames from each project's *Bugzilla* and *CVS* to a *MySQL* database. We then used these data to compute all the features as listed in Table 2. Note that we computed the features on the file level and for each of the available time frames (1, 2, 3, ... months) backwards from the prediction (target) point in time.

Most features' names are self-explanatory but some may need some additional context: The `activityRate` represents how many activities (revisions) took place per month. We include `grownPerMonth`, which describes the evolution of the overall project as a feature (in terms of lines of code). `chanceRevision` and `chanceBug` features describe the probability of having a revision and a bug in future akin to *Bug Cache* [11]. We compute those two features using the formula $1/2^i$, where i represents how far back (in months) the latest revision or bug occurred from the prediction time point. If the latest revision or bug occurrence is far from the prediction time point, then i is large and the overall probability of having a bug (or revision) in the near future is low. Hence, these variables model the assumption that files with recent bugs are more likely to have bugs in the future than others (see [4]). `LineOperIRTolLines` represents how many lines were added or deleted to fix a bug in relation to the total number of lines added / deleted. This indicates how much work is currently being done for fixing bugs in relation to other activities (such as adding new features).

3.2. Performance Measures

For most of our experiments we used class probability estimation (CPE) models. In our case the CPE model is a simple decision tree, which computes the probability distribution of a given instance over the two possible classes: `hasBug` and `hasNoBug`. Typically, one then chooses a cut-off threshold to determine the actual predicted class, which in turn can be used to derive a confusion matrix and accuracy. The problem of the accuracy as a measure is that it does not relate the prediction to the prior probability distribution of the classes. This is especially problematic in heavily skewed distributions such as the one we have (the ratio between defective files and non-defective ones is, depending on the project about 1:20 and approximately remaining this ratio in all samples). Therefore, we used the receiver operating characteristics (ROC) and the area under the ROC curve (AUC), which relate the true-positive rate to the false-positive rate resulting in a measure insensitive to the prior (or distribution) [16]. An AUC close to 1.0 is a good, one close to 0.5 represents a random prediction quality.

For the regression experiments we use linear regression models. The linear regression is a form of regression analysis in which the relationship between one or more independent

Name	Description
revision	Number of revisions
activityRate	Number of revisions per month
grownPerMonth	Project grown per month
totalLineOperations	Total number of line added and deleted
lineOperationRRevision	Number of line added and deleted per revision
chanceRevision	likelihood of a revision in the target period computed using $1/2^i$
lineAdded	# of lines added
lineDeleted	# of lines deleted
blockerFixes	# of blocker type bugs fixed
enhancementFixes	# of enhancement requests fixed
criticalFixes	# of critical type bugs fixed
majorFixes	# of major type bugs fixed
minorFixes	# of minor type bugs fixed
normalFixes	# of normal type bugs fixed
trivialFixes	# of trivial type bugs fixed
blockerReported	# of blocker type bugs reported
enhancementReported	# of enhancement requests reported
criticalReported	# of critical type bugs reported
majorReported	# of major type bugs reported
minorReported	# of minor type bugs reported
normalReported	# of normal type bugs reported
trivialReported	# of trivial type bugs reported
p1-fixes	# of priority one bugs fixed
p2-fixes	# of priority two bugs fixed
p3-fixes	# of priority three bugs fixed
p4-fixes	# of priority four bugs fixed
p5-fixes	# of priority five bugs fixed
p1-reported	# of priority one bugs reported
p2-reported	# of priority two bugs reported
p3-reported	# of priority three bugs reported
p4-reported	# of priority four bugs reported
p5-reported	# of priority five bugs reported
lineAddedI	# of lines added to fix bugs
lineDeletedI	# of lines deleted to fix bugs
totalLineOperationsI	Total number of lines operated to fix bugs
chanceBug	Likelihood of a bug in the target period computed using $1/2^i$
lineOperationIRbugFixes	Average number of lines operated to fix a bug
lineOperationIRTotalLines	# of lines operated to fix bugs relative to total line operated
lifeTimeBlocker	Average (avg.) lifetime of blocker type bugs
lifeTimeCritical	avg. lifetime of critical type bugs
lifeTimeMajor	avg. lifetime of major type bugs
lifeTimeMinor	avg. lifetime of minor type bugs
lifeTimeNormal	avg. lifetime of normal type bugs
lifeTimeTrivial	avg. lifetime of trivial type bugs
hasBug	Indicates the existence of a bug (Target Feature)

Table 2. File features

variables and another variable, called the dependent variable, is modeled by a least squares function, called a linear regression equation. This function is a linear combination of one or more model parameters, called regression coefficients. We report Pearson correlation, root mean squared error (RMSE), and mean absolute error (MAE) to measure the performance of the regression models.

4. Experiments: Showing the influence of Concept Drift

In this section we provide empirical evidence regarding the existence of *concept drift* in our four projects. We first show that the defect prediction quality changes over time. Then, in the second experiment we expand on this finding of variability and clearly visualize the periods of stability versus change indicating the existence of concept drift. The third subsection attempts to identify the features relevant for detecting *concept drift*. In other words we try to distill early-warning signs that (i) can be used to caution the usage of results from a bug-prediction model and (ii) might help to unearth the causes for the concept drift.

4.1. Defect prediction quality varies over time

The goal of this experiment is to show that the defect prediction quality varies over time. To that end we employ our features to learn a bug-prediction CPE model for each project. Specifically, we employ Weka's [17] J48 decision tree learner (a re-implementation of C4.5 [18]). To illustrate the large variation of prediction quality over time we trained on data preceding the target month (called the training period), predicted the number of bugs in the target month (or target period), and computed the AUC as a prediction quality measure. For example, if the initial target period is February, 2008, then the initial learning period is January, 2008. We then expanded the training period backwards in time by adding additional data (e.g., from December 2007) from the project's history. Depending on the length of the observation period for a project we could look back up to 74 months for Eclipse and Netbeans, 82 months for Mozilla, and 85 for Open Office. Next, we repeat the procedure by moving the target period one month back and use the preceding periods as training periods. We then visualize the prediction quality (AUC) of each model over time using a heat map (Figure 1 represents Eclipse. We had to omit the other three figures due to space considerations. However, they also exhibit similar characteristics as Eclipse). In the figure the X-axis indicates the target period and the Y-axis the length of the training period (in terms of number of months in past considered). Firstly, it is interesting to observe that in Figure 1, in some periods the model obtains high AUC while others are not. Also, we can see that in some prediction periods, initially the prediction quality is not so impressive but after expanding the learning period up to certain months back the model recovers the prediction quality. However, we can see in some cases that further expansion of learning period from that point could cause degradation of prediction quality. Lastly, it is interesting to observe that once a model has attained a certain accuracy adding additional older information will not destroy it. This indicates that the latest, predictive information is dominating in prediction [4]. The above features

can also be observed in the other three projects.
Summarizing, we clearly show that the defect prediction quality varies over time. This indicates that evaluating a model on one target period or only a few time points is not sufficient. Actually, choosing an optimal target and learning periods can convert a bad prediction model into a usable one and vice versa.

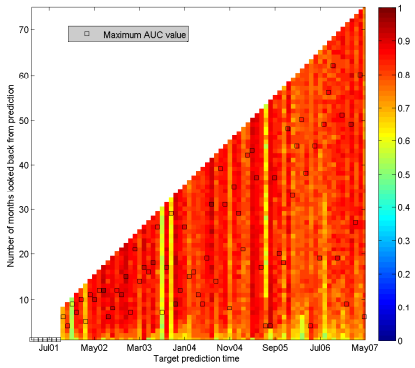


Figure 1. Eclipse: Historical heat-map with the point of highest AUC highlighted

4.2. Finding periods of stability and drift

So far we have seen that the prediction quality clearly varies over time. But are there clear periods of stability and drift (or change)? To clearly differentiate periods of stability and drift we slightly adapted our experiment as follows. Rather than training from the month directly preceding the target period and varying the length of training period we maintain the training period length constant (at 2 months) and move this time window into the past of the project. For example, if the initial target period is February 2008, then the initial training period is December 07 and January 08, followed by a period from November 07 and December 07 *etc*. [4], [1]. We use a 2-month learning window because the typical release cycle of the considered projects is usually 8 to 10 weeks. In addition, our previous work [1] has showed that 2 months of history data attains higher prediction quality. We use Weka's J48 decision tree and we measure the prediction quality using AUC as our first experiment. Again we assign a color for each AUC value and represent it in the heat maps (Figures 2, 3, 4, and 5). Note that whilst the X-axis of these graphs shows the target period as before, the Y-axis has a different meaning: it represents the more recent of the two months used for building the model. Hence, the higher in the figure we are looking the older is the two-month period compared to the target. Values on the diagonal (bottom left to top right) from each other represent predictions of the model trained on the same period.

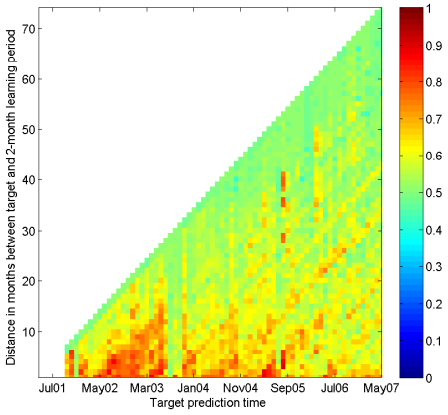


Figure 2. 2-month Heat-map: Eclipse

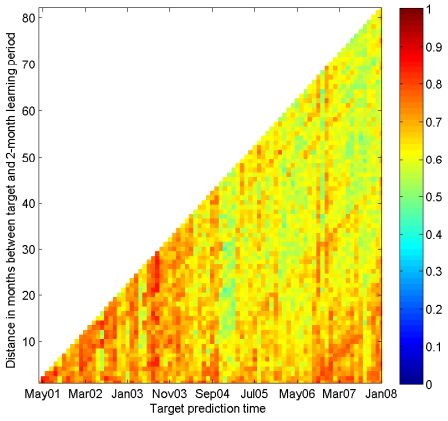


Figure 3. 2-month Heat-map: Mozilla

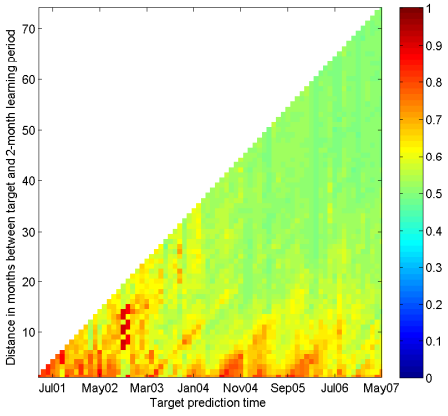


Figure 4. 2-month Heat-map: Netbeans

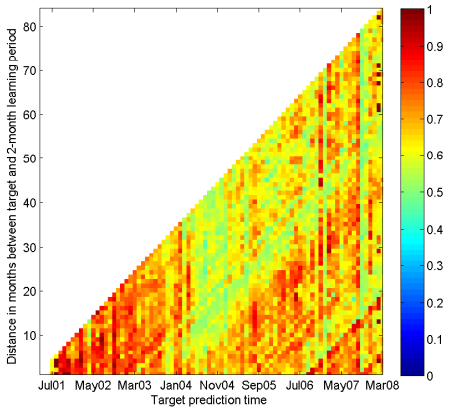
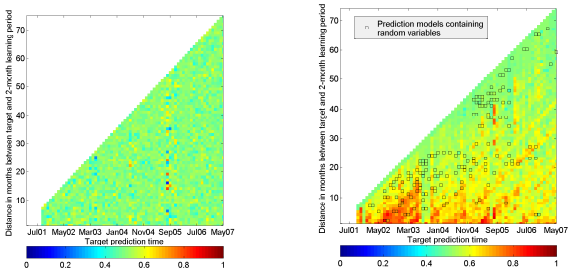


Figure 5. 2-month Heat-map: Open Office

Figure 2 clearly shows different triangle-shapes (red color). One triangle starts from April 2002 and continue to July 2003. In this time period the defect prediction quality is stable at an impressive AUC > 0.8. But suddenly, in August 2003, the defect prediction quality drops to almost random (AUC ≈ 0.5). Hence, the above period is so stable that even models learned on older data (in the summer of 2003 the model trains on data that is older than a year!) have excellent predictive power. This stability results in the triangle shape as the old training (along the upper left boundary/diagonal of the triangle) remains predictive. Also we can observe in all figures that further we go into the past the prediction quality drops down to almost random (≈ 0.5); proves the statement *the further you go in time the worst will be the prediction* [15]. More formally, from April 2002 to July 2003 the concept (*i.e.,* rules underlying the bug generation process and described partially by our features) remains consistent and, hence, the defect prediction quality is stable. Due to the concept drift in August 2003, the defect prediction quality drops down. In January 2004 the project seems to recover some stability and generate another, but slightly less pronounced triangle until November 2004. We can observe the similar effects in NetBeans with much shorter periods of stability and Open Office. In Mozilla this effect seems to be less pronounced maybe as we are really dealing with a set of subprojects. To illustrate that the triangle shapes are not an epiphenomenon of the data or the prediction algorithm, we also graphed the result of a naïve model that simply assumes that the defects of the learning period will be carried over to the target period. As Figure 6(a) clearly shows for Eclipse, most predictions attained in this manner are random (*i.e.,* AUC ≈ 0.5; green in the figure) and do not exhibit the triangle shapes. To prove that the triangles indeed visualize a phenomenon of the underlying data rather than the prediction process itself, we added 10 random variables to our feature set. We then tracked if these variables get picked for a prediction model by the algorithm. Figure 6(b) shows that random variables mostly get included in the model when the AUC is near 0.5 (*i.e.,* close to random). Only a few models inside the triangles contain random variables. Due to space considerations we had to omit the similar figures for the other projects.



(a) Naïve prediction model on Eclipse data

(b) Usage and position of random variables (Eclipse)

Figure 6. Experiments to exclude the possibility of the triangles being an epiphenomenon of the data or the prediction algorithm.

Summarizing, the model clearly exhibits periods of stability and periods of drift. The causes of the drifts – be they observable in our features or not – are not obvious from the graphs and will be investigated in the next subsection. Another interesting observation in the heat-maps is the height of the triangle-shapes. It indicates the length of the stable period. Note, that the height varies both within and between projects. Hence, an universal optimal training period length can not be determined but is highly dependent on the current stable period. Finally, this finding clearly indicates that decision makers in software project should be cautious to base their decisions on a defect prediction model. Whilst they might be useful in periods of stability they should be ignored in periods of drift. In the next experiment we investigate if these periods can be identified from the features we gathered to serve as early warning indicators with regard to the usage of defect prediction models.

4.3. Predicting periods of stability and drift

In the last experiment we show that defect prediction models exhibit periods of stability and drift. But can we uncover features that can be used to predict the kind of period that a software project is in to serve as indicators with regard to the usage of defect prediction models? To that end we learned a regression model to predict the AUC of the bug prediction model according to the following procedure: First, we computed the AUC of the bug prediction model based on the learning period in the two months before the target period in exactly the same way as in the previous subsection.

Second, since the AUC is a project-level feature of the prediction model we needed project level features to learn the prediction model. Thus, we computed a series of project level features that are listed in Table 7 for each target period. Third, since (i) the AUC prediction model used a 2 months training period and (ii) we are interested in changes between the training and the target period we transformed the features by taking the average of the two training months ($avg_t = average(feature_{t-1}, feature_{t-2})$) and subtracting it from the value of the target month ($= feature_t - avg_t$). Fourth and last, we build a traditional linear regression model predicting the AUC from these transformed features. The resulting regression models are shown in Tables 3, 4, 5, and 6. Note that if a regression coefficient is large compared to its standard error, then it is probably different from zero. The P-value of each coefficient indicates whether the coefficient is significantly different from zero such that if it is less than or equal to 0.05, then those variables significantly contribute to the model, else there is no significant contribution of those variables. The performance of the models is measured in terms of their Pearson correlation, mean absolute error (MAE), and root mean square error (RMSE) as in Table 8. Note that all models have a strong correlation between the predicted and actual values of AUC. Furthermore, the small MAE and RMSE reflect the good performance of our regression models.

Feature	Coefficient	P-value
(Constant)	0.67	0.000
enhancementFixes	0.0002	0.000
enhancementReported	0.0001	0.004
p1-fixes	−0.0013	0.000
p3-fixes	−0.0002	0.000
p5-fixes	−0.043	0.001
p1-reported	0.0015	0.000
p2-reported	0.0001	0.000
p3-reported	−0.0001	0.023
p4-reported	−0.0005	0.000
p5-reported	−0.005	0.000
LineOperationsIRbugFixes	−0.001	0.000
LineOperIRTolLines	−0.1127	0.000
author	−0.0065	0.000

Table 3. Eclipse: Regression Model

Feature	Coefficient	P-value
(Constant)	0.7333	0.000
revision	−0.0001	0.000
bugFixes	0.0001	0.000
enhancementFixes	−0.0012	0.000
enhancementReported	−0.0004	0.000
p1-fixes	0.0004	0.000
p2-fixes	0.0003	0.000
p3-fixes	0.0003	0.000
p4-fixes	0.0012	0.000
p5-fixes	−0.0016	0.000
p3-reported	0.0007	0.000
p4-reported	0.0005	0.001
p5-reported	0.001	0.000
LineOperationsIRbugFixes	0.0011	0.000
LineOperIRTolLines	−0.2478	0.000
author	−0.0007	0.001

Table 4. Mozilla: Regression Model

Feature	Coefficient	P-value
(Constant)	0.67	0.000
bugFixes	−0.0025	0.000
enhancementFixes	−0.0022	0.015
patchFixes	−0.002	0.01
featureFixes	−0.0024	0.000
enhancementReported	−0.0001	0.000
patchReported	0.0001	0.024
p2-fixes	0.0005	0.004
p2-reported	0.0025	0.038
p4-reported	0.0022	0.000
p5-reported	0.0035	0.000
LineOperIRTolLines	−0.0491	0.000
author	−0.0008	0.103

Table 5. Open Office: Regression Model

Feature	Coefficient	P-value
(Constant)	0.602	0.000
enhancementFixes	0.00027	0.000
patchFixes	0.004	0.000
featureReported	−0.0006	0.000
p4-fixes	−0.0001	0.035
p5-fixes	0.0024	0.000
p1-reported	−0.0001	0.000
LineOperIRTolLines	0.026	0.102
author	−0.0007	0.000

Table 6. Netbeans: Regression Model

In all regression models the change in the *number of authors* feature has a negative impact for the AUC. I.e. if the number of authors in the target period is larger than the number of authors in the learning period then the defect prediction quality goes down. Hence, the addition of new authors to a project will reduce the applicability of the defect prediction model learned without those authors. Adding new authors could be a cause for concept drift reminiscing the "don't add people to a late project" advice from Fred Brooks' Critical Man Month [12]. The regression coefficients for `author` in all four models are relatively small, but since the AUC moves in the range of $0.5 - 1.0$ they contribute about 1% to the model providing at least a qualitative indication.

Another interesting feature of the models is `LineOpeIRTotL`: number of lines added / removed to fix bugs relative to total number of lines operated. This feature reflects the fraction of work performed to fix bugs relative to total work done. In all of these models this factor has high impact on the models, since it has the highest coefficient. In Eclipse, Mozilla, and Open Office, this factor contributes negatively to the model, while in Netbeans it contributes positively. The higher this value, the more bugs are fixed in the next version, the lower the more new

Name	Description
revision	Number of revisions
grownPerMonth	Project grown per month
totalLineOperations	Total number of line added and deleted
bugFixes	Total number of bugs fixed in every type
bugReported	Total number of bugs reported in every type
enhancementFixes	Number of enhancement requests fixed
enhancementReported	Number of enhancement requests Reported
p1-fixes	# of priority one bugs fixed
p2-fixes	# of priority two bugs fixed
p3-fixes	# of priority three bugs fixed
p4-fixes	# of priority four bugs fixed
p5-fixes	# of priority five bugs fixed
p1-reported	# of priority one bugs reported
p2-reported	# of priority two bugs reported
p3-reported	# of priority three bugs reported
p4-reported	# of priority four bugs reported
p5-reported	# of priority five bugs reported
lineAddedI	# of lines added to fix bugs
lineDeletedI	# of lines deleted to fix bugs
totalLineOperationsI	Total lines operated to fix bugs
lineOperationIRbugFixes	Average number of lines operated to fix a bug
lineOperationIRTotalLines	Number of lines operated to fix bugs relative to total line operated
lifeTimeIssues	Average lifetime of all types bugs
lifeTimeEnhancements	Average lifetime of enhancement type bugs
authors	Total number of authors
workload	Average work done by an author
AUC	Area under ROC curve (Target)

Table 7. Project features

Project	pearson correlation	MAE	RMSE
Eclipse	0.59	0.046	0.061
Mozilla	0.57	0.045	0.057
Netbeans	0.65	0.041	0.056
Open Office	0.55	0.066	0.083

Table 8. Performance of the regression models

features are introduced. Hence, if the coefficient is negative as in Eclipse, Mozilla, and Open Office, then more new features are added than bugs fixed presumably leading to some stability with regards to bugs. Further we can support for the above statement since the Netbeans project has the smallest bug fixing rate per file (0.32) compared to the other three projects (Eclipse: 0.43, Mozilla: 3.36 and Open Office: 0.94).

One important issue to note is that whilst `LineOpeIRTotL` contributes strongly to the Netbeans model, it does not do so significantly ($p = 10.2\%$). One could, therefore, hypothesize that in Netbeans, in contrast to the other projects, most bugs are fixed by experienced authors whose behavior is well captured by the model. To test this proposition we computed the fraction of work done by the authors, who are not in the learning period but in target period, to fix bugs. Figure 7 graphs the result for one target period (the others are omitted due to space considerations), where the X-axis represents time into the past from the target period and the Y-axis represents the fraction of bug fixing performed by new authors. The figure clearly shows that in Eclipse and Mozilla most of bugs are fixed by those authors, who are not in the learning period and the fraction continuously increases the further we look back into past. In Open Office the fraction of work done by new authors drastically varies and is probably not meaningful due to a significantly smaller number of transactions (commits) per month. For Netbeans the fraction of work done by new authors to fix bugs is initially very small and does never rise above about 50% with a mean well below 40%. Also, the number for Netbeans is relatively constant indicating some stability in its developer base. Hence, mostly experienced authors seem to be fixing bugs increasing the models prediction quality as those authors behavior is already known in the learning period.

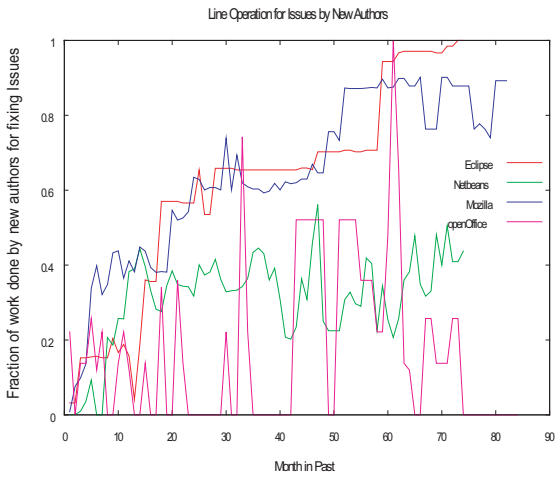


Figure 7. Work done by new authors to fix bugs

Note that the feature `enhancementFixes` is occupied by all four regression models. However, this feature is not consistent since in Open Office and Mozilla it contributes negatively while in Netbeans and Eclipse it is positive. Therefore, it is difficult to figure out the behavior of this feature in the context of software engineering.

Summarizing, we observed that rising the number of authors editing the project could cause the drop of the defect prediction quality. We also saw that more work done to fix bugs relative to the other activities as well causes a reduction of the defect prediction quality. Therefore, the behavior of these two features could be considered as an early warning signal for *concept drift*.

Exploring author fluctuations: The above observations encouraged us to further investigate the relationships between

author fluctuation, bug fixing activity, and stable versus drift periods. To that end we identified tipping points from stable to drift periods in each of the projects and graphed the normalized change in number of authors and normalized change in bug fixing activity for the months preceding the onset of the drift and some months into the drift. Consider Figure 2 as an example, the "stable" months leading up to the tipping month of August 2003 and including the "drifting" month of October 2003. The value for the authors are computed as shown below.

$$\frac{\#auth_{month} - \#auth_{month-1}}{\sum_{t \in months} |\#auth_t - \#auth_{t-1}|}$$

In words, the difference between the number of authors ($\#auth$) of the month and its preceding month normalized by the sum of the differences of all the months considered in the graph. The value for changes in bug fixes is computed analogously. The rationale for the normalization is to make the figures somewhat comparable across different projects and time-frames.

Figures 8, 9, 10, 11, and 12 show a selection of the resulting figures, which are titled by the "tipping" month.

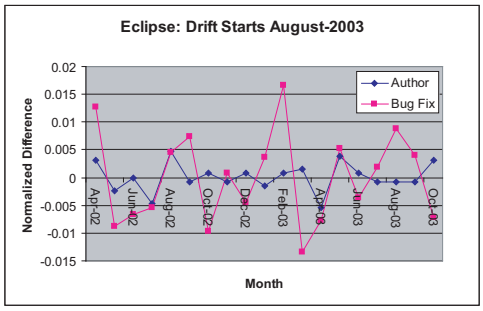


Figure 8. Eclipse: Drift starts in August 2003

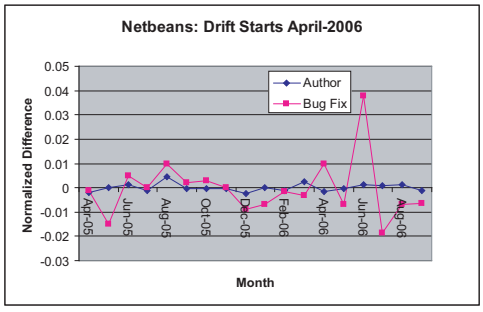


Figure 9. Netbeans: Drift starts in April 2006

All five figures show a relative drop in authors before or in the "tipping" month mostly followed by an increase in authors during the drift. We also find that in most cases, the relative amount of work done for bug fixing increases massively in the first month of the drift. Unfortunately,

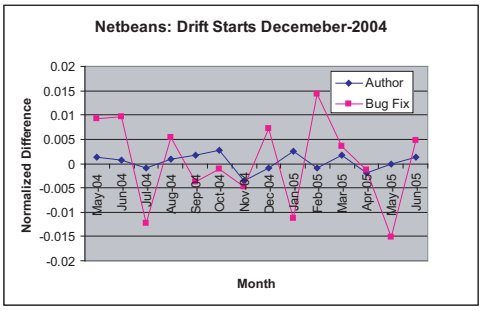


Figure 10. Netbeans: Drift starts december 2004

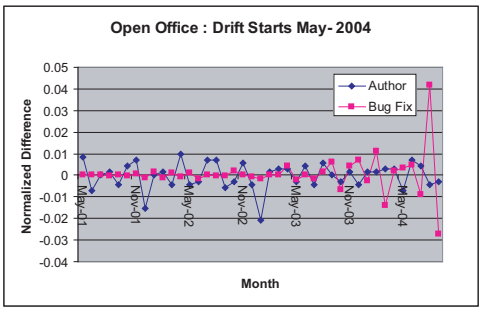


Figure 11. Open Office: Drift starts in May 2004

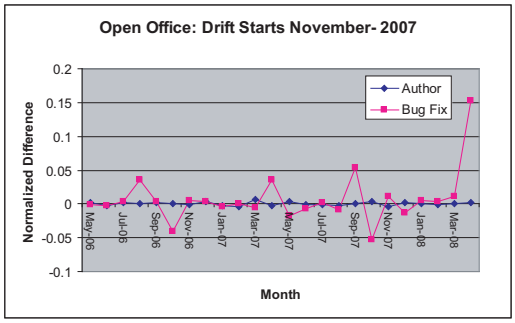


Figure 12. Open Office: Drift starts in November 2007

neither of these observations is unique to the tipping periods. Considering Eclipse (Figure 8), *e.g.*, we find that normalized author differential tips 3 times: in January 03, April 03, and preceding the drift in July 03. The same can be said for the normalized bug differential. Hence, we cannot argue that these factors can be used exclusively to predict periods of drift, but together they can serve as a basis for developing such an early warning indicator.

Summarizing, this third experiment shows that the prediction of drift periods seems to be possible and pursuing early warning indicators for drifts seems to be a promising endeavor. In addition, the results highlight that author / developer fluctuations as well as changes in the amount of work expended to fix bugs in relation to adding new features seem to *correlate* with changes in prediction quality.

From a software engineering standpoint these correlations can definitely be explained and would uphold some time-honored principles.

5. Conclusions and Future Work

This paper investigated the notion of *concept drift* in data from software projects. We were specifically interested in drifts of the concept "bug generation process" as it would impact defect prediction algorithms. Using data from four open source projects we found that the quality of defect prediction approaches indeed varies significantly over time. We, furthermore, found that the quality of the prediction clearly follows periods of stability and drift, indicating that concept drift *is indeed an important factor to consider* when investigating defect prediction. As a consequence, *the benefit of bug prediction in general must be seen as volatile over time and, therefore, should be used cautiously*.

In a further experiment we attempted to uncover the underlying causes of *concept drift* in a software project. We observed that number of authors editing the project is rising right before, or during a *concept drift*. This reinforces the well-known software engineering rule "adding manpower to a late software project makes it later"[12]. We also saw a relationship between the changes of the proportion of work done to fix bugs and other activities and the defect prediction quality. Unfortunately, both those correlations were not observed uniformly in connection with *concept drift* and can only serve as a start to elicit early warning indicators for *concept drift* and, hence, the reduced quality of existing defect prediction models. We plan to further investigate the question about the causes of *concept drift* in software projects. In the ideal case it would be possible to identify *the* influential factors that hold for software projects in general. Whatever the outcome of our future investigations, we can safely say that the notion of *concept drift* seems to have a profound influence in the empirical investigation of software evolution.

References

[1] A. Bernstein, J. Ekanayake, and M. Pinzger, "Improving defect prediction using temporal features and non linear models," in *Proceedings of the International Workshop on Principles of Software Evolution*, 2007.

[2] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[3] T. Khoshgoftaar, E. Allen, N. Goel, A. Nandi, and J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system," in *Proceedings of the 7th International Symposium on Software Reliability Engineering*, 1996.

[4] A. Hassan and R. Holt, "The top ten list: dynamic fault prediction," in *Proceedings of the 21st International Conference on Software Maintenance*, 2005.

[5] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.

[6] A. Tsymbal, "The problem of concept drift: Definitions and related work," Department of Computer Science Trinity College, Tech. Rep., 2004.

[7] A. Mockus and L. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings of the International Conference on Software Maintenance*, 2000.

[8] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005.

[9] P. Knab, M. Pinzger, and A. Bernstein, "Predicting defect densities in source code files with decision tree learners," in *Proceedings of the 2006 international workshop on mining software repositories*. ACM, 2006.

[10] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering*. IEEE Computer Society, 2007.

[11] S. Kim, T. Zimmermann, E. J. W. Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007.

[12] F. P. Brooks and F. Phillips, *The mythical man-month: essays on software engineering*. Addison-Wesley Reading, MA, 1995.

[13] M. Harries and K. Horn, "Detecting concept drift in financial time series prediction using symbolic machine learning," in *Proceedings of the 8th Austrailian Joint Conference on Artificial Intelligence*. World Scientific Publishing, 1995.

[14] G. Widmer and M. Kubat, "Effective learning in dynamic environments by explicit context tracking," in *Proceedings of the European Conference on Machine Learning*, 1993.

[15] B. Kenmei, G. Antoniol, and M. D. Penta, "Trend analysis and issue prediction in Large-Scale open source systems," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, 2008.

[16] F. Provost and T. Fawcett, "Robust classification for imprecise environments," *Machine Learning*, vol. 42, no. 3, 2001.

[17] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[18] J. R. Quinlan, *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., 1993.

Time variance and defect prediction in software projects

Jayalath Bandara Ekanayake
Dept. of Physical Sciences & Technology
Faculty of Applied Sciences
Sabaragamuwa University of Sri Lanka
Belihuloya
Sri Lanka

Tel: 0094 711954433
Email: jayalathek@sab.ac.lk

CURRICULUM VITAE

EDUCATION

University of Zurich, Switzerland
***Doctoral Program in Informatics* (2012 April)**
Dissertation: " Improving Reliability of Defect Prediction Models: From Temporal Reasoning and Machine Learning Perspective
Advisor: Prof. Dr. Abraham Bernstien (PhD)
Co-advisor: Prof. Dr. Harald C. Gall

University of Peradeniya, Sri Lanka
***MSc* in Computer Science (2003 September)**
Dissertation: "Identifying M16 Land Mines using Backpropagation Neural Network"
Advisor: Dr. J. Wijekulasooriya

University of Peradeniya, Sri Lanka
***BSc* (Hons) in Physical Science (1997 March)**

TEACHING EXPERIENCE

Sabaragamuwa University of Sri Lanka
Lecturer (Probationary) –June 01, 2004 to to-date

Sabaragamuwa University of Sri Lanka
Instructor (Computer Technology) –June 01, 1999 to May 31, 2004

University of Zurich, Switzerland
Teaching Assistant/ Research Assistant- December 08, 2006- January 10, 2011

Sri Lanka Military Academy, Diyathalawa
Visiting Lecturer- 2001 to 2006

University of Peradeniya, Sri Lanka
Demonstrator (Computer Technology) -1997 to 1999

PUBLICATIONS

Ekanayake, J., Tappolet, J., Gall, H., and Bernstein, A. (2011). Time variance and defect prediction in software projects. Empirical Software Engineering, pages 1–42. 10.1007/s10664-011-9180-x.

Ekanayake,J., Tappolet,J., Gall,H., Bernstein,A., Time variance and defect prediction in software projects: Towards an exploitation of periods of stability and change as well as a notion of concept drift in software projects: additional figures, Version: 2, 2011. (Technical Report)

Ekanayake, J., Tappolet, J., Gall, H. C., and Bernstein, A. (2009). Tracking Concept Drift of Software Projects Using Defect Prediction Quality. In Proceedings of the 6th IEEE Working Conference on Mining Software Repositories. IEEE Computer Society.

Bernstein, A., Ekanayake, J., and Pinzger, M. (2007). Improving defect prediction using temporal features and non linear models. In IWPSE '07: Ninth international workshop on Principles of software evolution, pages 11–18, New York, NY, USA. ACM.

Reviews

"Preserving Knowledge in Software Projects". Journal of Systems and Software (2011).

"Computer aided length estimation of Coconut fiber". Indian Journal of Science and Technology (2011).

"Human Capital Value and Uniqueness, Human Resource Management Practices and Human Resource Configurations: A Study of Software Development Firms". ITRU Research Symposium 2011, University of Moratuwa (2011).

"Drive Moderator methods for Retail Prediction". International Journal of Information Technology and Decision Making (2012).

"Sportscast: Investigating the Possibility of Replacing Television Sports Broadcasting With Live Internet Broadcasting". Ceylon Journal of Science (Physical Sciences) (2012).

"An efficient algorithm for line clipping in computer graphics programming". Ceylon Journal of Science (Physical Sciences) (2012).

"Ontology Driven Clinic Management for Sri Lankan General Hospitals", M.Phil thesis (2013 February), Postgraduate Institute of Science, University of Peradeniya

Research Interests

- Mining Software Repositories
- Machine Learning
- Pattern Recognition

Current Research Projects

- Modeling Software Quality: From the Perspective of Machine Learning and Data Mining.
- Effective Algorithms for Resource Allocation.
- Query Answering System in Agricultural Domain.