View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Waterloo's Institutional Repository

Personalized Defect Prediction

by

Tian Jiang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2013

© Tian Jiang 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Academia and industry expend much effort to predict software defects. Researchers proposed many defect prediction algorithms and metrics. While previous defect prediction techniques often take the author of the code into consideration, none of these techniques build a separate prediction model for each developer. Different developers have different coding styles, commit frequencies, and experience levels, which would result in different defect patterns. When the defects of different developers are combined, such differences are obscured, hurting the prediction performance.

This thesis proposes two techniques to improve defect prediction performance: *personalized defect prediction* and *confidence-based hybrid defect prediction*. Personalized defect prediction builds a separate prediction model for each developer to predict software defects. Confidence-based hybrid defect prediction combines different models by picking the prediction from the model with the highest confidence. As a proof of concept, we apply the two techniques to classify defects at the file change level. We implement the state-of-the-art change classification as the baseline and compare with the personalized defect prediction approach. Confidence-based defect prediction combines these two models. We evaluate on six large and popular software projects written in C and Java—the Linux kernel, PostgreSQL, Xorg, Eclipse, Lucene and Jackrabbit.

Acknowledgements

I would like to take this opportunity to express my greatest gratitude to my supervisor, Prof. Lin Tan. It is her that provided me with the chance to work at the frontier of the software reliability research. During the past two years, I received countless invaluable advice from her. I am trained in every aspect such as research insights, technical skills, time management, presentation skills and collaborations. The two-year experience working with her will benefit me in any position that I would take in the future.

I am extremely grateful to Prof. Sung Kim from the Hong Kong University of Science and Technology who provided me with helpful insights during the development of my project. I am very appreciated that the readers, Prof. Patrick Lam and Prof. Mahesh V. Tripunitara, would spare the time from their very busy schedules to review my thesis and provide precious comments. I am thankful to all of our research group members. It is a pleasant time working and discussing with them. I want to thank the University of Waterloo, and the Department of Electrical and Computer Engineering, for providing the environment and back-end supports.

Lastly, I would also like to thank my parents for supporting me through my education career. Their support made it possible to study in the first-tier university.

This thesis and my graduate career are not possible without the help from any of these people.

Table of Contents

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Personalized Change Classification 2

1.2 Confidence-based Hybrid Change Classification 2

1.3 Multivariate Adaptive Regression Splines (MARS) 3

1.4 Metrics and Results 3

1.5 Contributions 4

2 Related Work 5

3 Change Classification (CC) 7

3.1 Buggy Change Labelling 7

3.2 Feature Extraction 8

3.2.1 Characteristic Vector 8

3.2.2 Bag-of-Words 9

3.2.3 Meta-Data 10

3.3 Classification and MARS 10

4 Personalized Change Classification (PCC) 11

5 Combining CC and PCC (PCC+) 14

6 Experimental Setup 16

6.1 Research Questions 16

6.2 Test Subjects 17

6.3 Data Set Setup 18

6.4 Classification and Tuning 19

6.5 Measures 20

6.6 Statistical Tests 21

7 Experimental Results 22

7.1 PCC Versus CC 22

7.2 Combining PCC and CC 25

7.3 Generalizability of PCC's Improvement 26

7.4 Effect of Different Features 30

8 Discussion and Future Work 33

8.1 Cost Effectiveness vs. F1 33

8.2 Interpretation of Defect Prediction Results 34

8.3 Threats to Validity 34

9 Conclusions 36

References 37

List of Tables

6.1 Evaluated projects. The numbers in this table include only C/Java code. [a]Xserver subrepository. [b]Eclipse JDT Core subrepository. 17

6.2 The setup of the data set, showing start gaps, start dates, end dates and the buggy rate in the data sets. The buggy rate is the overall buggy rate for both PCC and CC since they use exactly the same data sets. 18

7.1 Results of evaluated projects. The values in parentheses show the F1, NofB20 and PofB20 differences against CC. The "Average" row contains the average improvement of PCC over CC across all projects. Statistically significant improvements are bolded. 23

7.2 F1 and NofB20 for different classifiers. The delta between CC and PCC are shown in the "Delta" row. The "Average" row contains the average delta between PCC and CC across all projects for each classification algorithm. For simplicity, the p-values are not shown. Instead, statistically significant deltas are bolded. . . 26

7.3 The effect of different classes of features on F1. M represents meta-data features. B represents bag-of-words features. C represents characteristic vector features. The values in parentheses show the deltas against the predictions using only meta-data features. For simplicity, the p-values are not shown. Instead, statistically significant deltas are bolded. 30

7.4 The effect of different classes of features on NofB20. M represents meta-data features. B represents bag-of-words features. C represents characteristic vector features. The values in parentheses show the deltas against the predictions using only meta-data features. For simplicity, the p-values are not shown. Instead, statistically significant deltas are bolded. 31

List of Figures

3.1 The characteristic vector for the above code segment contains one `if` statement, two `for` loops and zero `while` loops. 9

4.1 Buggy rates (percentages of buggy changes) of different syntactic structures of four Linux kernel developers. The syntactic structures are modulo operators (`%`), `for` loops, bit-wise `or` operators (`|`), and `continue` statements. Developers have different buggy change patterns, which cannot be observed if we combine different developers' changes ("total"). 12

5.1 The flow diagram of PCC+. First, CC, PCC and weighted PCC predict the changes and pass the confidences to the meta classifier. Then, the meta-classifier picks the highest confidence. Last, the converter converts the highest confidence to a prediction (buggy or clean). 15

7.1 Cost effectiveness graphs that show the percentage of bugs that can be discovered by inspecting different percentages of LOC. PCC is better than CC for a wide range of LOC choices. 24

7.2 F1 versus number of training changes for all test subjects. X-axis is the number of training changes for CC, and 1/10 of it is the number of training changes per developer for PCC. 27

7.3 PofB20 versus number of training changes for all test subjects. X-axis is the number of training changes for CC, and 1/10 of it is the number of training changes per developer for PCC. 28

7.4 NofB20 versus number of training changes for all test subjects. X-axis is the number of training changes for CC, and 1/10 of it is the number of training changes per developer for PCC. 29

8.1 These three ADTree nodes are from three developers' trees. The first node means that the weight of a change being buggy increases by 0.815 if developer dev1 removes any bit-wise `and` operator in this change. A change is predicted buggy if the weight is positive. 34

Chapter 1

Introduction

Academia and industry have spent a great effort in predicting software defects [5, 19, 21, 23, 31, 43, 44, 45, 71, 72]. These prior studies made significant advances in defect prediction using various features such as code complexity, code locations, the amount of in-house testing, historical data, and socio-technical networks.

This thesis summarizes our experience in improving change classification performance. We propose a novel technique, personalized change classification (PCC), which builds a separate model for each developer. We also propose an enhancement, confidence-based hybrid change classification (PCC+), which combines personalized change classification and traditional change classification.

This thesis can also serve as a guide on how to improve experiment performance. This thesis demonstrates the effect of different classification algorithms, adding or removing features, and the number of training instances. Although prior work all argue briefly about why they set up the data set in a certain way, none of them fully demonstrate the effect of different setups.

In this thesis, we will use the following terminology. A *commit* is a transaction unit in a source code management system, e.g., Git. A *change* contains the modifications to one file in a commit. A *classifier* and a *classification algorithm* both refer to a machine learning algorithm, e.g., linear regression. We use these two terms interchangeably. A classifier learns from the *features*, which quantitatively describe the changes. The property to be predicted is called *label*. In this thesis, we use binary labels, i.e., *clean* and *buggy*. Clean means the code is correct. Buggy means the code contains bugs. The process of learning patterns from a data set is called *training*. The data set used in training is called the *training set*, whose labels are given to the classifier. After training, the classifier produces a *model*. One can use the model to predict on future data. The data set used to evaluate the model is called the *test set*. The test set's labels are not given

to the classifier. We compare the prediction results against the true labels to evaluate the model. *Defect prediction* refers to the technique that collects features about some subjects, learns using classified data and predicts on unclassified data. *Change classification* refers to the change-level defect prediction as described by Kim et al. [31].

1.1 Personalized Change Classification

While many of the previous defect prediction studies take the author of the code into consideration, none of these studies build separate prediction models for individual developers. They combine all developers' changes to build a single prediction model.

Different developers have different coding styles, commit frequencies, and experience levels, all of which cause different defect patterns [53]. For example, based on our inspection of the Linux kernel's mainline repository from 2005 to 2010, 48.0% of one developer's changes related to `for` loops are buggy while the percentage is only 13.3% for another developer. When the defects of different developers are combined, such differences are obscured, hurting prediction performance. Therefore, it is desirable to build personalized defect prediction models. Analogously, search engines such as Google use personalized search to capture the different search patterns to provide improved search experience [18].

In this thesis, we propose *personalized defect prediction*—building separate prediction models for individual developers to predict software defects. As a proof of concept, we apply our personalized defect prediction idea to change classification [31], which we call *personalized change classification (***PCC***)*. Change classification predicts defects at the change level. In the rest of this thesis, we refer to the approach that builds a single change classification model for all developers as the *traditional change classification (***CC***)* approach.

1.2 Confidence-based Hybrid Change Classification

Many classification algorithms, such as ADTree [16], produce models that can provide a confidence measure for each prediction result. We can use the confidence measure to compare the confidence of different predictions on the same change. For example, the confidence measure of model A predicting on one change is 0.8. It means that we are confident about model A's prediction to a level of 0.8. On the other hand, if the confidence measure of model B predicting the same change is 0.9, it is wise to follow model B's prediction.

Based on the above observation, we propose ***PCC+*** to further improve prediction performance. PCC+ augments PCC with other models, e.g., CC, by picking the prediction result from the model with the highest confidence for each change. Similar to PCC, PCC+ is also capable of producing developer-specific models.

1.3 Multivariate Adaptive Regression Splines (MARS)

Similar to PCC, a recent study by Bettenburg et al. [5] breaks data into clusters and builds separate prediction models for different clusters. Bettenburg et al. applies *Multivariate Adaptive Regression Splines (MARS)* [17] to defect prediction, which is a global model approach that takes local considerations into account. In contrast to our approach that groups data by developers, MARS groups data to minimize the fitting error. Bettenburg et al. [5] shows that MARS outperforms the traditional global model approach of building a single classification model from the entire data set. Therefore, we compare our personalized classification models against MARS models in addition to comparing our personalized classification models against the traditional global models. This work is further discussed in the next chapter, Related Work.

1.4 Metrics and Results

To evaluate the proposed approaches, we use two widely used metrics: *Cost Effectiveness* [3, 58, 59] and *F1-score* [31, 59]. Rahman et al. [59] pointed out that cost effectiveness is a suitable measure for evaluating the performance of defect predictions in cost sensitive scenarios. The cost effectiveness evaluates the classification performance given the same cost, which is typically measured by the lines of code (LOC) to inspect. For example, when a team can afford to inspect only 20% lines of code before a deadline, it is crucial to inspect the 20% that can help the developers discover the most number of bugs. In this thesis, we use the same cost effectiveness measure from Rahman et al. [59] with a small variation: the number of bugs that can be discovered by inspecting 20% LOC (***NofB20***). Both Rahman's and our measures use the same cost effectiveness graph. Rahman's measure considers the area under the curve for the percentage of LOC from 0 to 20%. While Rahman's measure can ensure a high average between 0 and 20%, our measure is much easier to interpret. If PCC improves NofB20 by 100 compared to CC, it means that PCC can help the developers identify 100 more bugs by inspecting the top 20% LOC identified by PCC instead of the top 20% LOC identified by CC. We also present cost effectiveness graphs so that developers can view the number of bugs that can be discovered by inspecting percentages of LOC other than 20%. In addition, we evaluate the approaches on the standard

F1-score, which is also widely used in defect prediction [31, 59]. F1-score is an appropriate performance measure when there are enough resources to inspect all predicted buggy changes, e.g., code inspection. If a change is predicted buggy, the developers can put more testing and verification effort into this change. A higher F1-score can help capture more bugs and reducing the time wasted on inspecting true clean changes.

We evaluate the proposed techniques on six large and popular projects written in C and Java: the Linux kernel, PostgreSQL, Xorg, Eclipse, Lucene and Jackrabbit. PCC improves NofB20 by up to 155 and 187 where CC and MARS can discover 55 and 20, respectively. PCC also improves F1-score by 0.01–0.06 and 0.01–0.13 compared to CC and MARS, respectively. Statistical tests show that the above improvements are all statistically significant. In addition, we show that the improvements are not limited to a specific experimental setup, i.e., the classification algorithm and the number of changes in the training set.

1.5 Contributions

In summary, this thesis makes the following contributions:

- We propose the personalized change classification (PCC) to improve the traditional change classification (CC).
- We propose PCC+, which combines PCC and CC, to further improve prediction performance.
- Our evaluation demonstrates that PCC and PCC+ outperform CC and MARS in terms of both cost effectiveness and F1-score.
- We demonstrate the effect of several experimental setup parameters.

Chapter 2

Related Work

To the best of our knowledge, we are the first to build a separate model for each individual developer for defect prediction. In this section, we focus on discussing other differences and connections between this thesis and the previous work.

Many studies [19, 21, 23, 31, 40, 43, 44, 45, 58, 72] analyze the effects of code complexity, process metrics, code locations, the amount of in-house testing, historical data, socio-technical networks, etc., to build prediction models and predict software defects. Kim et al. [31] use support vector machine (SVM) as the classifier; bag-of-words, complexity metrics and metadata as the features to predict defects. Recently, Shivaji et al. [62] improve this work by applying feature selection algorithms. Ostrand et al. [53] use negative binomial regression as the classifier; various metadata and developer-specific metrics as features to predict defects. Rahman et al. [58] compare the effect of code metrics and process metrics. Lumpe et al. [40] investigate the importance of activity-centric static code metrics. These studies advanced the state of art of defect prediction. However, none of them builds personalized models although some of them have taken the developers as an important feature. We notice the importance of the difference between the developers, and improve the change classification performance by building personalized models.

The personalized idea has great success in other fields. For example, Google's personalized search can enhance search results by leveraging the user's search history [18]. Facebook and MySpace's personalized advertisement deployment can predict the user's interests by analyzing the user's profile [66]. We introduce the personalized idea to the change classification field. Our results show that the personalized approach can improve change classification performance.

Bettenburg et al. [5] compares the performance of three models: a local model that consists of several linear models, a single global linear model and a global model that has local considerations, MARS. These three models are learned from the same data set. The local model first

clusters the data set based on Euclidean distance, and then runs linear regression on each cluster. The global model runs linear regression on the whole data set. MARS is a technique that clusters the data set to minimize the overall fitting error, while running linear regression in each cluster. Bettenburg et al. find that MARS is better than both the global model and the local model. Compared to the global model, MARS divides data into clusters and fit one linear model for each cluster, which can describe non-linear relationships more accurately than a single linear model. Compared to the local model, MARS's clustering strategy is to minimize global fitting error. The local model clusters the data set based on Euclidean distance, which may still not be accurately described by local linear models. Although MARS can group data into clusters to minimize fitting error, it does not group data by developer. We instead propose a developer-specific local model approach that utilizes domain specific knowledge and outperforms MARS. Chapter 4 further discusses the differences between our personalized approach and MARS.. Furthermore, we propose a hybrid technique that can take advantage of different models.

Matsumoto et al. [41] find that developer-related metrics are good distinguishing factors for defect prediction. Specifically, the modules that are touched by more developers contain more bugs. Rahman and Devanbu [57] find that a developer's experience on one file is more important than the developer's general experience on the project. Posnett et al. [55] introduce focus metrics and find that more focused developers introduce fewer bugs. These papers propose several developer related features that have strong correlation with the probability of being buggy. In addition to traditional features [31], such as meta-data and bag-of-words, we use a new type of features—characteristic vector [27]. In the future, we may add these developer related features to improve our defect prediction.

Rahman et al. [59] argue that defect prediction should also consider the cost effectiveness metrics. They show that two predictions can be statistically indistinguishable by looking at the F1-score, but statistically different by looking at the cost effectiveness metrics. The F1-score and cost effectiveness metric emphasize different aspects. A discussion about the difference is in Chapter 8. Herzig et al. [24] show that misclassified bug reports have a great impact on defect prediction. They manually labeled some bug reports and show that using manually labeled data and automatically labeled data can yield significantly different top error-prone file lists. This thesis follows these learned lessons. In addition to the standard F1-score, we report the number of bugs that can be discovered by examining 20% of code, a.k.a, NofB20. Two test subjects in this thesis use manually labeled data from the work of Herzig et al. [24].

Chapter 3

Change Classification (CC)

This section gives a brief overview of the traditional change classification technique (CC) and our adaptation of this traditional change classification technique..

The traditional change classification has the following steps:

1. Label each change *clean* or *buggy* by mining the project's revision history [14, 63] (Section 3.1).
2. Extract features, such as bag-of-words, from the changes (Section 3.2). To improve classification performance, we add a new type of features—*characteristic vectors*—to the traditional change classification for the first time. Characteristic vectors can capture some more information about the syntactic structures of the change compared to bag-of-words.
3. Use a classification algorithm to build a model from the labelled changes based on the extracted features (Section 3.3). Since MARS [17] has been shown as a superior classification algorithm [5], we use it as another baseline.
4. Predict new changes as buggy or clean using the model.

3.1 Buggy Change Labelling

We adopt the same definition of *change* as prior work [31]. Compared to defect prediction at the other levels, e.g., the file level, defect prediction at the change level is more precise. For a

predicted buggy change, the developer needs to examine only one change instead of the entire file to address the defect.

To label each change as buggy or clean, we follow the method used by previous works [14, 63]. We first identify *bug-fixing* changes—changes that fix bugs–by searching the commit logs for the word "fix". The lines that the bug-fixing changes modified are assumed to be the location of a bug. This approach has a precision of 75–88% in identifying bug-fixing commits in our data set (Section 8.3).

Modern source-code management systems provide an *annotate* functionality such as `git blame`, which annotates each line with the most recent change that modified that line. The changes that introduced those buggy lines are *bug-introducing* changes (also referred to as *buggy* changes).

In addition to the method described above, we also pick two projects from the work of Herzig et al. [24], who manually verified the bug reports to distinguish real bugs from feature enhancements. If a bug report contains a real bug, its associated changes are bug-fixing changes. These two projects have high quality commit messages that contain bug report IDs. Instead of searching for the word "fix", we search for the bug IDs of the real bugs in the commit messages while labeling bug-fixing changes [63]. This approach has a precision of 91–95% in identifying bug-fixing commits in these two projects (Section 8.3). We then use the same method to label bug-introducing changes.

3.2 Feature Extraction

Features are attributes extracted from a change which describe the characteristics of the source code, such as LOC.

We use three classes of features: 1) characteristic vectors, 2) bag-of-words, and 3) meta-data. **This thesis does not intend to find the best feature combination; instead it compares personalized prediction models against traditional prediction models, using the same features for both models**. We briefly study the effect of different classes of features. The results are in Section 7.4.

3.2.1 Characteristic Vector

Inspired by the Deckard tool [27], we use characteristic vectors as features. Characteristic vectors represent the syntactic structure by counting the numbers of each node type in the Abstract

```
// Sum up positive entries
for (int i = 0; i < array.length; ++i) {
  for (int j = 0; j < array[i].length; ++j) {
    if (array[i][j] > 0) sum += array[i][j];
  }
}
```

Figure 3.1: The characteristic vector for the above code segment contains one `if` statement, two `for` loops and zero `while` loops.

Syntax Tree (AST). Bag-of-words (see Section 3.2.2) and characteristic vectors have different abstraction levels. Although bag-of-words can capture keywords, such as `if` and `while`, it cannot capture abstract syntactic structures, such as the number of statements.

Suppose we are using `if`, `for`, and `while` node types for characteristic vectors. The code shown in Figure 3.1 has a characteristic vector of (1, 2, 0). The characteristic vectors can be calculated by recursively summing these vectors while traversing the AST. Depending on the syntax definition, the AST might have different representations of the syntax, but the basic idea remains the same. For each change, we automatically generate two characteristic vectors: one for the source code file before the change and one for the source code file after the change. After getting the characteristic vectors for the file before the change and the file after the change, we subtract the two characteristic vectors to obtain the difference. For example, for a change that removes one `for` loop, the difference is (0, -1, 0). We use the difference between the two characteristic vectors and the characteristic vector of the file after the change as two sets of characteristic vector features.

3.2.2 Bag-of-Words

Defects that are caused by calling a wrong function, such as `malloc` instead of `calloc`, cannot be represented by characteristic vectors, because characteristic vectors ignore the identifier names. To address this issue, we add the bag-of-words (BOW) [60] vectors as features. It converts a string to a word vector of individual words, where each entry of the vector is the corresponding occurrence of each word.

We use a Weka [20] filter with the Snowball [1] stemmer to convert text strings to word vectors. We process both the commit message and the source code to obtain the word vector for each change.

3.2.3 Meta-Data

In addition to characteristic vectors and bag-of-words features, we also use meta-data features. We collect developer, commit hour (0, 1, 2, ..., 23), commit day (Sunday, Monday, ..., Saturday), cumulative change count, cumulative buggy change count, source code file/path names, and file age in days in a way similar to that of Kim et al. [31]. Since the developer is a feature in CC, PCC does not take advantage of the extra knowledge of the developer information.

3.3 Classification and MARS

Given the labels (buggy or clean) and the features extracted from the source code files, classification algorithms can build models and predict new buggy changes. We use off-the-shelf classification algorithms from Weka [20].

Since PCC and CC are two different approaches to organize data sets for defect prediction, and neither is tied to any specific classification algorithm, we compare PCC and CC using three widely used [8, 39, 58, 71, 73] classification algorithms: Alternating Decision Tree (ADTree) [16], Naive Bayes [28] and Logistic Regression [37]. This thesis does not intend to find the best-fitting classifiers or models, but to **compare PCC models against CC models using the same classification algorithms**.

In addition to CC, we use Multivariate Adaptive Regression Splines (MARS) as another baseline for the following reasons. First, similar to our personalized defect prediction approach, MARS builds separate models for different groups of data to improve the classification performance. Second, MARS performs better than both global models and local models in defect prediction [5].

We use the MARS implementation in the off-the-shelf machine learning framework Orange [12]. The underlying MARS implementation is the same as used by Bettenburg et al. [5].

Chapter 4

Personalized Change Classification (PCC)

Different developers have different experience levels, different coding styles, and different commit patterns, resulting in different buggy change patterns [53]. This chapter describes Personalized Change Classification (PCC).

Figure 4.1 shows differences in buggy change patterns of four Linux kernel developers in the mainline repository from 2005 to 2010. Specifically, it shows the buggy rates (percentage of buggy changes) of four syntactic structures for each developer. The syntactic structures are modulo operators (`%`), `for` loops, bit-wise `or` operators (`|`), and `continue` statements. The x-axis presents different developers and the y-axis indicates the buggy rate for each syntactic structure. The last group "total" shows the sum of all four developers' changes. Each bar shows the ratio of a developer's buggy changes regarding one syntactic structure to the developer's changes regarding the same syntactic structure. For example, 48% of developer c's 320 changes adding or removing `for` loops are buggy.

We observe that these developers have different buggy change patterns: (1) while developer b's modulo operators (`%`) are much buggier than developer c's, developer b's other syntactic structures are cleaner than developer c's; (2) developer a's buggy rate is consistently lower than that of the other developers; and (3) the buggy rates of the same syntactic structure for different developers are different. These different patterns cannot be observed if we sum the four developers' changes as shown by "total".

Given such differences in developers' buggy changes, a prediction model built from a developer's changes alone can be more suitable for predicting the changes from the same developer. Therefore, we build a separate prediction model for each developer. PCC follows exactly the same steps as CC except that PCC groups changes by developers and builds a separate model

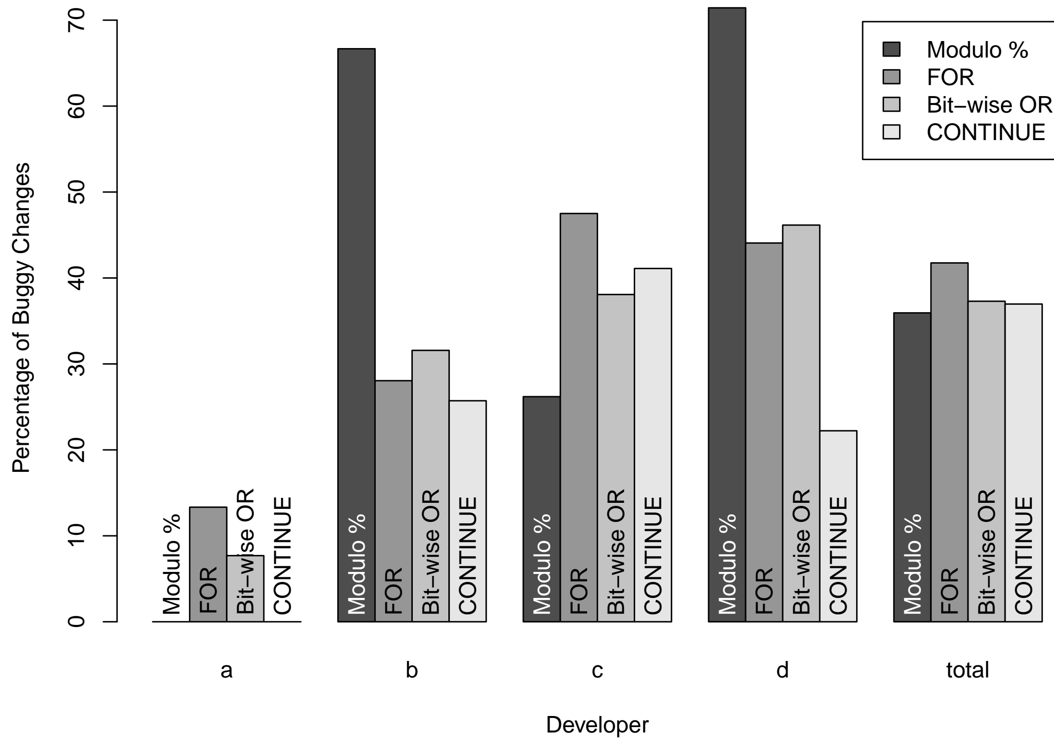


Figure 4.1: Buggy rates (percentages of buggy changes) of different syntactic structures of four Linux kernel developers. The syntactic structures are modulo operators (`%`), `for` loops, bit-wise `or` operators (`|`), and `continue` statements. Developers have different buggy change patterns, which cannot be observed if we combine different developers' changes ("total").

for each developer. Given a new unlabelled change, the final model first checks the author of the change, and then uses this developer's model to predict this change.

Note that **the idea behind PCC is not adding the feature—developer—as an advantage over CC, but instead building separate models for different developers.** In fact, the developer is a meta-data feature used in our implementation of CC. PCC's main advantage over CC is that each developer's model can be better tailored for this developer to capture the developer's unique buggy change patterns.

Compared to MARS, Our personalized change prediction approach is different in two aspects: (1) while MARS partitions data to minimize fitting errors, our PCC approach chooses to partition data by developers, based on our observation that different developers have different development behaviors and patterns; and (2) while a MARS model is a global model that takes

local considerations into account, our PCC model is a local model, because the individual model for each developer is learned from a subset of the data. In summary, our PCC approach is a specialized local model approach that utilizes domain specific knowledge.

Chapter 5

Combining CC and PCC (PCC+)

To further improve classification performance, we take advantage of both PCC and CC. We propose two enhancements which combine PCC and CC— *weighted PCC* and *PCC+*.

Firstly, a PCC model for a single developer may overlook common bug patterns across developers, which can be learned from the other developers within the project. We enhance the PCC models by adding the changes from the other developers to a developer's model. We call this approach *weighted PCC*. We use the same process as PCC except that we use different training sets. Recall that in PCC, all changes in one developer's training set are collected from this developer's changes. In weighted PCC, we collect half of the changes from one developer, and the other half from all other developers. Note that this is different from CC because the weight (i.e., number of changes) of one developer's changes in the training set is higher in the weighted PCC approach.

Secondly, we automatically pick the prediction with the highest confidence among CC, PCC, and weighted PCC. We refer to this approach as *PCC+*. Many classification algorithms, such as ADTree [16], can provide a *confidence* measure. The confidence measure tells us how sure the model is about a prediction. The flow diagram of PCC+ is shown in Figure 5.1. For PCC+, we predict each change using these models (CC, PCC, and weighted PCC) independently, and pick the prediction with the highest confidence for each change as the final prediction. An alternative approach to PCC+ is majority voting. We choose the confidence-based approach because majority voting is likely to yield an incorrect prediction when the majority is not confident.

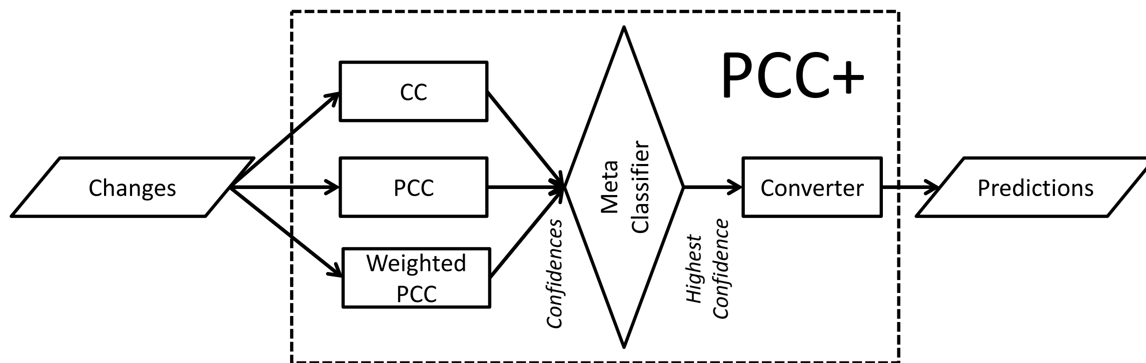


Figure 5.1: The flow diagram of PCC+. First, CC, PCC and weighted PCC predict the changes and pass the confidences to the meta classifier. Then, the meta-classifier picks the highest confidence. Last, the converter converts the highest confidence to a prediction (buggy or clean).

Chapter 6

Experimental Setup

6.1 Research Questions

We design our experiments to investigate the following research questions (RQs):

RQ1. How much do PCC and PCC+ improve the classification performance over CC and MARS?

First, we compare PCC with the previous approaches, namely CC and MARS. We find that **PCC outperforms both CC and MARS in terms of both NofB20 and F1** (Section 7.1). Second, we investigate whether PCC+ can further improve classification performance. We find that **it is feasible to combine prediction models based on the confidence measure to further improve classification performance** (Section 7.2).

RQ2. Is PCC's performance gain over CC generalizable to other experimental setups?

We investigate whether the improvement of PCC is only specific to the experimental setup in our evaluation. First, we investigate if the performance improvement of PCC over CC is generalizable to other classification algorithms. We evaluate PCC and CC on three classification algorithms. We find that **PCC significantly outperforms CC with all evaluated classification algorithms** (Section 7.3).

Second, we investigate the effect of the number of training instances. Since each PCC model learns from a training set that is smaller than that of CC, PCC may not yield a good performance with few changes per developer in smaller projects while CC may still perform well. We investigate how many training instances are enough for PCC to yield a better performance than CC. **In**

Table 6.1: Evaluated projects. The numbers in this table include only C/Java code. [a]Xserver subrepository. [b]Eclipse JDT Core subrepository.

Project	Language	LOC	First Commit Date	Last Commit Date	# of Changes	% of Buggy Changes
Linux	C	7.3M	2005-04-16	2010-11-21	429K	14.0%
PostgreSQL	C	289K	1996-07-09	2011-01-25	89K	23.6%
Xorg[a]	C	1.1M	1999-11-19	2012-06-28	46K	12.9%
Eclipse[b]	Java	1.5M	2001-06-05	2012-07-24	73K	16.9%
Lucene	Java	828K	2010-03-17	2013-01-16	76K	9.4%
Jackrabbit	Java	589K	2004-09-13	2013-01-14	61K	23.6%

general, PCC models learned from 80 training instances or more per developer can yield a better performance than CC (Section 7.3).

RQ3. What is the difference between using different combinations of feature classes?

There are three feature classes, i.e., bag-of-words, characteristic vectors and meta-data. We investigate the effect of using different combinations of feature classes. We find that **the difference between different feature classes are too small to justify the cost** (Section 7.4).

6.2 Test Subjects

We choose six open-source projects: the Linux kernel, PostgreSQL, Xorg, Eclipse, Lucene and Jackrabbit. These projects have enough change history to build and evaluate PCC, and they are commonly used in the literature [13, 14, 24, 31, 71]. For Lucene and Jackrabbit, we use manually verified bug reports from Herzig et al. [24] for labeling bug-fixing changes, and the keyword searching approach [63] for the others.

Table 6.1 shows detailed project information. The lines of code (LOC) and the number of changes in Table 6.1 include only source code (C and Java) files[1] and their changes because we want to focus on classifying source code file changes. Our subjects vary from 289K to 7.3M in terms of LOC and from 46K to 429K in terms of the number of changes. They are large and typical open source projects covering operating system, database management system and core applications.

[1]We include files with these extensions: .java, .c, .cpp, .cc, .cp, .cxx, .c++, .h, .hpp, .hh, .hp, .hxx and .h++.

Table 6.2: The setup of the data set, showing start gaps, start dates, end dates and the buggy rate in the data sets. The buggy rate is the overall buggy rate for both PCC and CC since they use exactly the same data sets.

Project	Start Gap	Start Date	End Date	% of Buggy Changes
Linux	Three Years	2008-01-23	2008-07-15	21.0%
PostgreSQL	Two Years	1998-07-08	2010-02-14	40.9%
Xorg	Three Years	2003-07-02	2009-07-24	23.1%
Eclipse	Three Years	2004-06-07	2006-01-24	23.0%
Lucene	Six Months	2010-09-17	2011-06-30	31.0%
Jackrabbit	Three Years	2007-09-13	2009-09-15	46.4%

Although these projects are written in C and Java, PCC is not limited to any particular programming language. With appropriate feature extraction, PCC could classify changes in any language [31].

6.3 Data Set Setup

We use the entire revision histories to label buggy changes as described in Section 3.1. After labeling buggy changes, we choose changes from a certain period for the following reasons. First, there are too many changes, and often classification algorithms such as MARS do not scale to a large number of instances. In addition, as shown in previous work [14, 11, 32], the average bug life time varies from one year to three years depending on the project. Therefore, the latest changes are often labelled clean even if they are actually buggy simply because the bugs in these changes have not been discovered and fixed. To ensure that we have all necessary fixes to label the buggy changes, we exclude the changes in the last three years. Third, there is also a concern that the change patterns may not be stable at the beginning of the histories [31, 22]. For this reason, we exclude the changes at the beginning of the histories.

Table 6.2 shows start gaps, start dates and the average buggy rates of the data sets. The start and end dates are the dates of the first and last commits in the data sets. The start gaps are three years for most of the projects. Lucene and PostgreSQL are two special cases with different start gaps. Since Lucene has a relatively short history (three years), the start gap is six months, and we do not remove any history at the end. PostgreSQL has less than five core developers. The

required time span to collect enough changes for a non-core developer is long. To collect enough changes, we choose a two-year start gap for PostgreSQL, and we remove the last year of history.

We set our data sets to perform a fair comparison between PCC and the baselines, i.e., CC and MARS. There are a few key requirements for the experimental setup: 1) we use a mixed data set for the baselines, but developer-specific data sets for PCC; 2) we combine the training sets of PCC to use as the training sets of CC; and 3) we keep the test data the same between PCC and the baselines. With these goals in mind, we select the ten developers from each project who have the most commits. They are the most prolific developers and can represent the majority of each project. We then pick the same number of changes from each of the developers to prevent any developer's performance from dominating. Furthermore, we use 10-fold cross-validation [46] to reduce the bias on the training set selection. This technique is widely used in the literature [5, 31, 33, 58]. PCC is personalized, so we run cross validation on the changes of one developer at a time. For the baselines, we run cross validation on all changes together.

In summary, we first list the developers in each project ordered by their total number of commits in descending order. From the list, we select the top ten developers. For each developer, we collect 100 consecutive changes starting after the start gaps, totaling 1,000 changes per project. Each 100 changes are used for PCC for each developer. The 1,000 changes together are used for CC and MARS.

6.4 Classification and Tuning

We design our tuning strategy carefully to simulate real world scenarios. The key problem is that the golden labels of the test set are unknown in practice. We cannot use the golden labels to tune parameters. Therefore, we need to tune the parameters on labeled data set before performing predictions in practice. The goal becomes finding one set of parameters that is appropriate for these experiments in general. Specifically, we tune on some randomly picked changes, and then use the same parameter for all models and all projects. We tune the parameters to maximize F1 since it is a standard metric.

In our experiments, we use four classification algorithms, including MARS. We tune each classification algorithm separately. The implementations of our Logistic Regression and Naive Bayes have no tunable parameters [20], so they are not tuned. ADTree has one parameter, the number of boosting iterations [20], so we use a linear search. MARS has several parameters that take arbitrary values, e.g., maximum degree, fast k, and penalty for hinges in the GCV computation [12]. Since each run of MARS can take hours, it is infeasible to explore the search space exhaustively. Instead, we tune one parameter first using a linear search, fix its value, tune the next parameter, and so on.

6.5 Measures

We use two performance measures for our evaluation: Cost effectiveness and F1. Different measures are useful in different scenarios. Cost effectiveness is useful when there are only resources to inspect a limited amount of code, e.g., before a deadline. F1 is useful when there is enough resource to inspect all predicted buggy changes, e.g., code review. A detailed discussion is in Chapter 8.

Cost Effectiveness: Cost effectiveness is a widely used metric for defect prediction [59, 58, 3]. As the name suggests, cost effectiveness aims at maximizing benefits by spending the same amount of cost. In this thesis, the cost is the amount of code to inspect, and the benefit is the number of bugs that can be discovered. If we inspect all predicted buggy changes, the percentage of bugs that we catch is the recall. In some situation, e.g., under deadline pressure, we cannot inspect all predicted buggy changes, instead we can inspect perhaps only 20% of committed LOC. It is desirable to catch as many bugs as possible while minimizing LOC to inspect. In this situation, the cost effectiveness metric is a more appropriate measure.

Although our prediction is binary (i.e., a change is either buggy or clean), there can be multiple bugs in a change. It is more appropriate to consider that we can capture multiple bugs after we inspect a change. We borrow the idea from the previous work [52, 59], but adopt it to the change level. Recall the technique that we use to label buggy changes: we use `git blame` on the bug-fixing changes. One bug-introducing change can be `git blame`d by multiple bug-fixing changes. We use the number of bug-fixing changes that `git blame`s a bug-introducing change as the number of bugs in this bug-introducing change, which are discovered by inspecting this bug-introducing change.

To evaluate the cost effectiveness, we rank the changes by the probability of being buggy, similar to previous work [58, 59]. We simulate the inspection process by looking at the changes sorted by their rankings. As we inspect the changes, we accumulate the inspected LOC and the discovered bugs. We plot the cost effectiveness graph with the accumulated inspected LOC on the x-axis and the accumulated discovered bugs on the y-axis. Although not restricted to any percentage, we use the number of bugs captured by inspecting 20% of committed LOC, i.e., ***NofB20***, as a quantitative measure similar to previous work [59]. Since different projects have different numbers of bugs, we also normalize NofB20 as a percentage of the total number of bugs, i.e., ***PofB20***. We use NofB20 and PofB20 as the quantitative metrics of cost effectiveness and use the cost effectiveness graph to show the general trend across different percentages of LOC.

F1: F1 is a standard and widely used [31, 71, 72] measure for classification algorithms, which is the harmonic mean of precision and recall. Precision represents among all the predicted buggy

changes, how many are truly buggy. Recall represents among all the actual buggy changes, how many the classifier identifies. Precision and recall are calculated from numbers of true positives (TP), false positives (FP), false negatives (FN) and true negatives (TN). If the actual label is buggy and the classifier says buggy, it is a true positive. If the actual label is clean but the classifier says buggy, it is a false positive. If the actual label is buggy but the classifier says clean, it is a false negative. If the actual label is clean and the classifier says clean, it is a true negative. Precision P is calculated by $P = TP/(TP + FP)$, and recall R is calculated by $R = TP/(TP + FN)$.

There is a trade-off between precision and recall. Usually, we can sacrifice one to improve the other, which makes it difficult to compare different prediction models using the precision alone or the recall alone.

6.6 Statistical Tests

Statistical tests can help us understand whether there is a statistically significant difference between two results that we want to compare[2]. For example, in RQ1, we want to compare the F1s of PCC and CC in each test subject. We first repeat each experiment several times to obtain several samples for each test subject. We then apply the Wilcoxon signed-rank test on the samples in each test subject and across subjects. The Wilcoxon signed-rank test does not require the underlying data to follow any distribution, can be applied on pairs of data, and is able to compare the difference against zero. At the 95% confidence level, p-values that are smaller than 0.05 indicates that the F1/NofB20/PofB20 differences between PCC and CC are statistically significant. p-values that are 0.05 or larger indicates that we find no evidence that the differences are statistically significant.

[2]We consult frequently with the statistical consulting service provided by the University of Waterloo, who monitors the experiments and ensures that we utilize the proper statistical tests correctly.

Chapter 7

Experimental Results

7.1 PCC Versus CC

In this section, we compare PCC against CC and MARS. We evaluate the three approaches using the setups introduced in Chapter 6, and measure cost effectiveness, precision, recall, F1, NofB20 and PofB20 as described in Section 6.5. In addition, we use statistical tests to check if result differences are statistically significant as described in Section 6.6.

Table 7.1 presents the overall classification performance of CC, MARS and PCC. The values in parentheses show the improvements of F1, NofB20 and PofB20 against CC. The "Average" row contains the arithmetic means of the improvements between PCC and CC across all projects. The statistically significant improvements are bolded. We use ADTree in RQ1 since decision tree is widely used in the literature [8, 58].

Cost Effectiveness: Table 7.1 shows that PCC improves CC by 19–155 in terms of NofB20 and by 0.03–0.21 in terms of PofB20. **All improvements are statistically significant**.

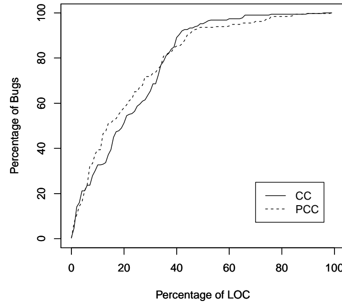
NofB20 represents the number of bugs that can be discovered by examining the top 20% LOC. For example, the ranking of CC can help the developers identify 55 bugs for PostgreSQL. On the other hand, the ranking of PCC can help identify 210 bugs, which is 155 more bugs than those of CC. PofB20 represents the same information but normalized to the number of bugs in the data set of a project. For example, PofB20 improvement on PostgreSQL is 0.21 means that the 155 bugs are 21% of all the bugs in PostgreSQL's data set.

The cost effectiveness graphs are in Figure 7.1. As shown in the figure, PCC is better than CC for a wide range of LOC choices other than 20%. For example, PCC and CC of Lucene diverge when the percentage of LOC is at about 5% and converge around 60%.

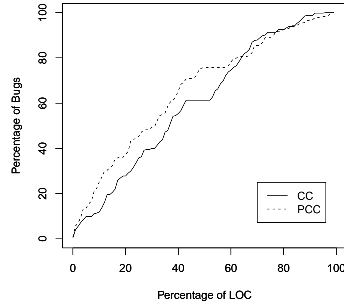
Table 7.1: Results of evaluated projects. The values in parentheses show the F1, NofB20 and PofB20 differences against CC. The "Average" row contains the average improvement of PCC over CC across all projects. Statistically significant improvements are bolded.

Project	Method	P	R	F1	NofB20	PofB20
Linux	CC	0.59	0.49	0.54	160	0.51
	MARS	0.46	0.39	0.42	121	0.39
	PCC	0.61	0.50	0.55(+0.01)	179(+**19**)	0.57(+**0.06**)
	PCC+	0.62	0.49	0.55(+0.01)	172(+**12**)	0.55(+**0.04**)
PostgreSQL	CC	0.65	0.58	0.61	55	0.08
	MARS	0.60	0.55	0.57	76	0.11
	PCC	0.63	0.58	0.60(−0.01)	210(+**155**)	0.29(+**0.21**)
	PCC+	0.66	0.59	0.63(+0.02)	175(+**120**)	0.24(+**0.16**)
Xorg	CC	0.69	0.62	0.65	96	0.23
	MARS	0.65	0.52	0.57	152	0.37
	PCC	0.69	0.66	0.67(+**0.02**)	159(+**63**)	0.39(+**0.16**)
	PCC+	0.73	0.66	0.69(+**0.04**)	161(+**65**)	0.39(+**0.16**)
Eclipse	CC	0.59	0.48	0.53	116	0.20
	MARS	0.55	0.43	0.48	20	0.03
	PCC	0.63	0.55	0.59(+**0.06**)	207(+**91**)	0.36(+**0.16**)
	PCC+	0.68	0.56	0.61(+**0.08**)	200(+**84**)	0.35(+**0.15**)
Lucene	CC	0.58	0.46	0.51	176	0.28
	MARS	0.51	0.41	0.45	131	0.21
	PCC	0.60	0.53	0.56(+**0.05**)	254(+**78**)	0.40(+**0.12**)
	PCC+	0.64	0.54	0.59(+**0.08**)	258(+**82**)	0.41(+**0.13**)
Jackrabbit	CC	0.72	0.72	0.72	411	0.37
	MARS	0.72	0.70	0.71	411	0.37
	PCC	0.72	0.72	0.72(+0.00)	449(+**38**)	0.40(+**0.03**)
	PCC+	0.74	0.74	0.74(+**0.02**)	459(+**48**)	0.41(+**0.04**)
Average				(+**0.03**)	(+**74**)	(+**0.12**)

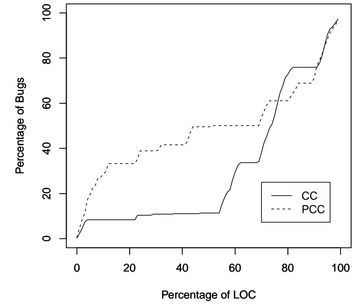
F1: As shown in Table 7.1, PCC improves CC by up to 0.06 on F1. For example, CC's F1 on Eclipse is 0.53. PCC's F1 on the same data set (Eclipse) is 0.59, which indicates that PCC improves the F1 by 0.06. The associated p-value is <0.01, which indicates that the improvement is ***statistically significant***. Although the improvement on PostgreSQL is negative, the associated p-values indicates that the difference is not statistically significant. We find no evidence that PCC



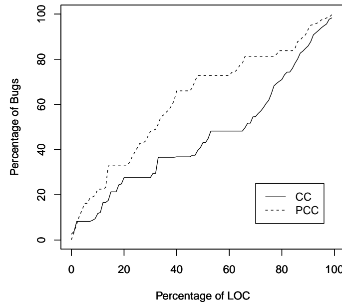
(a) The Linux kernel



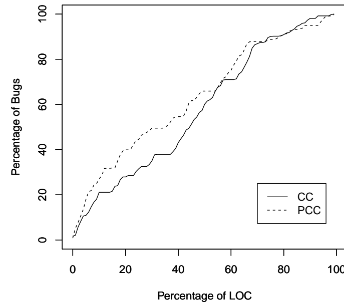
(b) Xorg



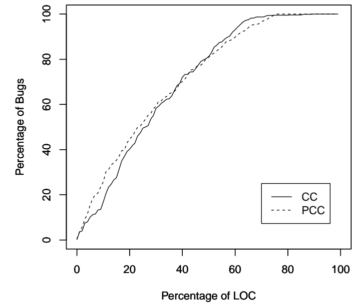
(c) PostgreSQL



(d) Eclipse



(e) Lucene



(f) Jackrabbit

Figure 7.1: Cost effectiveness graphs that show the percentage of bugs that can be discovered by inspecting different percentages of LOC. PCC is better than CC for a wide range of LOC choices.

and CC perform differently predicting PostgreSQL in terms of F1. Unlike the other test subjects, the PostgreSQL community requires the author field to reflect the committer. Since we do not have the information of the actual author, PCC's advantage may not show, which could affect the performance of PostgreSQL (Section 8.3). As shown in the "Average" row, **PCC outperforms CC on average across all test subjects**.

Overall, these promising results indicate that PCC can capture developers' different defect patterns by building personalized prediction models. The difference in defect patterns can be blurred when the changes from different developers are mixed together, which is a limitation of CC.

We have shown that PCC's F1 is higher than CC's. In addition, PCC has comparable or higher precision than CC for all projects. Some developers may prefer precision while others prefer recall. To address this issue, we can trade recall for precision and vice versa by simply tuning the classification algorithm parameters. Kim et al. [31] showed the trade-off between precision and recall.

MARS: In addition, we evaluate MARS on the same data sets for comparison. As shown in Table 7.1, PCC clearly outperforms MARS for all subjects in terms of both NofB20 and F1. For example, the F1 of Eclipse changes predicted by PCC is 0.59, while the F1 is only 0.48 when predicted by MARS on the same data set.

MARS's local consideration does not explicitly distinguish developers. Our result shows that PCC, a local model which utilizes domain specific knowledge, outperforms a global model with local consideration.

PCC outperforms both CC and MARS.

7.2 Combining PCC and CC

In this section, we compare PCC+ to CC and PCC. We use the same setup and subjects as Section 7.1.

PCC+ is a meta-classifier that picks the most confident result among CC, PCC and weighted PCC as described in Chapter 5. Recall that we use ADTree in RQ1. According to Freund et al. [16], an ADTree-based classifier can provide a confidence measure. For each change, we predict using the three models, i.e., CC, PCC and weighted PCC. PCC+ picks the result that has the highest confidence.

As shown in Table 7.1, PCC+ can further improve on PCC in many cases. For example, CC's F1 on Eclipse is 0.53. PCC improves CC by 0.06. PCC+ has a bigger improvement, 0.08. Interestingly, PCC+'s improvement on Jackrabbit's F1 is statistically significant, while PCC's is not. Compared to PCC, we observe that PCC+ is more likely to yield a statistically significant improvement and the improvement is often bigger.

Combining models based on the confidence measure can provide further improvements.

Table 7.2: F1 and NofB20 for different classifiers. The delta between CC and PCC are shown in the "Delta" row. The "Average" row contains the average delta between PCC and CC across all projects for each classification algorithm. For simplicity, the p-values are not shown. Instead, statistically significant deltas are bolded.

Project	Approach	F1			NofB20		
		ADTree	N. B.	L. R.	ADTree	N. B.	L. R.
Linux	CC	0.54	0.39	0.39	160	138	102
	PCC	0.55	0.40	0.49	179	147	137
	Delta	+0.01	**+0.01**	**+0.10**	**+19**	**+9**	**+35**
PostgreSQL	CC	0.61	0.51	0.56	55	89	46
	PCC	0.60	0.52	0.56	210	113	56
	Delta	−0.01	+0.01	+0.00	**+155**	**+24**	+10
Xorg	CC	0.65	0.55	0.63	96	84	52
	PCC	0.67	0.60	0.65	159	101	29
	Delta	**+0.02**	**+0.05**	**+0.02**	**+63**	**+17**	**−23**
Eclipse	CC	0.53	0.43	0.53	116	65	54
	PCC	0.59	0.47	0.51	207	108	55
	Delta	**+0.06**	**+0.04**	**−0.02**	**+91**	**+43**	+1
Lucene	CC	0.51	0.42	0.44	176	152	30
	PCC	0.56	0.45	0.50	254	139	200
	Delta	**+0.05**	**+0.03**	**+0.06**	**+78**	−13	**+170**
Jackrabbit	CC	0.72	0.56	0.72	411	420	261
	PCC	0.72	0.66	0.68	449	414	370
	Delta	+0.00	**+0.10**	**−0.04**	**+38**	−6	**+109**
Average	Delta	**+0.03**	**+0.04**	**+0.02**	**+74**	**+12**	**+50**

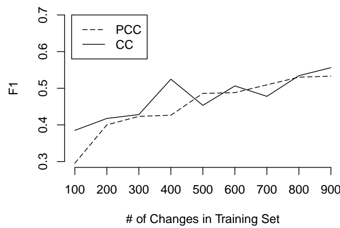
7.3 Generalizability of PCC's Improvement

In this section, we study the effect of different classification algorithms and different sizes of training sets.

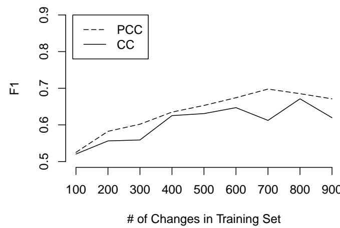
Different Classification Algorithms: We deploy three classification algorithms to check if PCC outperforms CC with other classification algorithms. We conduct this experiment using the same data sets in Section 7.1, but we vary the classification algorithm from just ADTree.

Table 7.2 shows the results of PCC and CC using various classifiers. The columns show the project name, the approach, the CC and PCC results with ADTree classifiers taken from Table 7.1, and the results with other classifiers—Naive Bayes and Logistic Regression. The results demonstrate that PCC outperforms CC for all classifiers, even though the performance of different classifiers varies. As shown in the "Average" row, the improvement of PCC is statistically significant for every classification algorithm. In other words, the benefit of grouping changes by developer (PCC) is not limited to a specific classification algorithm.

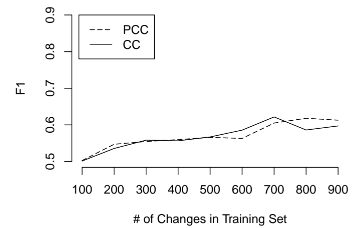
PCC outperforms CC, and this advantage is not limited one specific classification algorithm.



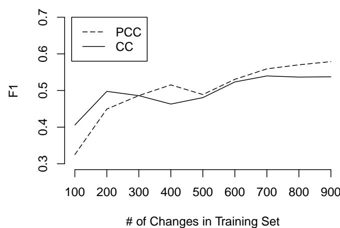
(a) The Linux kernel



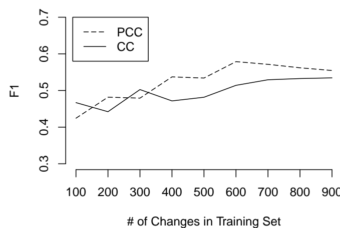
(b) Xorg



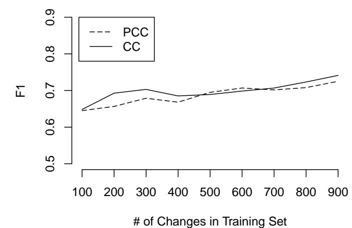
(c) PostgreSQL



(d) Eclipse



(e) Lucene



(f) Jackrabbit

Figure 7.2: F1 versus number of training changes for all test subjects. X-axis is the number of training changes for CC, and 1/10 of it is the number of training changes per developer for PCC.

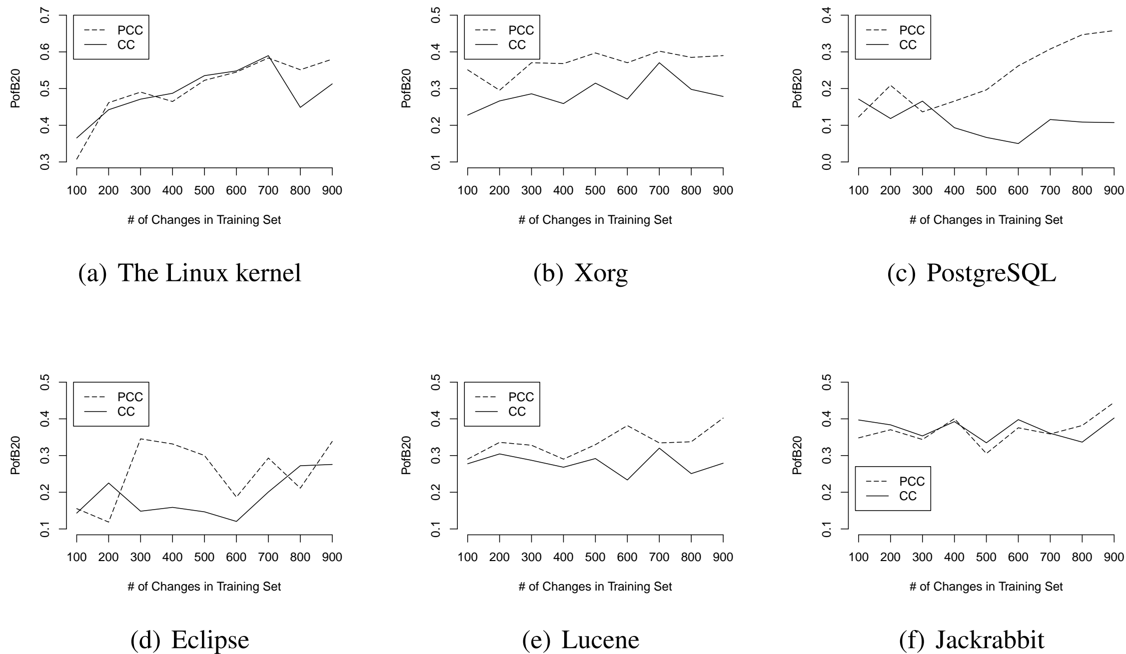
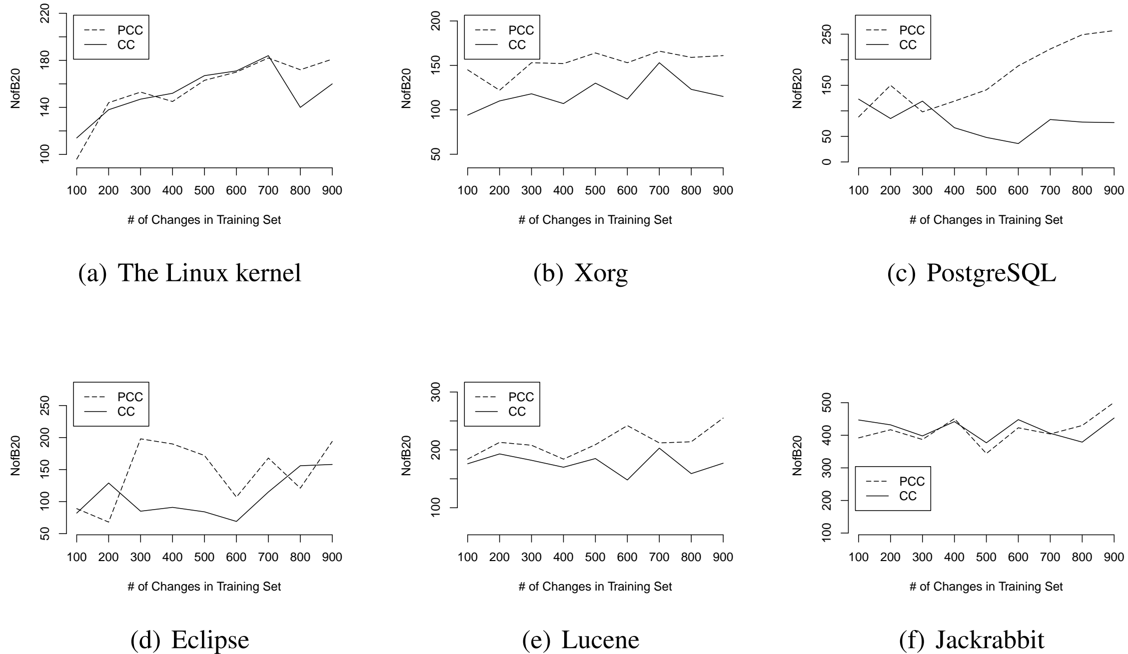


Figure 7.3: PofB20 versus number of training changes for all test subjects. X-axis is the number of training changes for CC, and 1/10 of it is the number of training changes per developer for PCC.

Number of Training Instances: Both CC and PCC learn from training instances. Unfortunately, it is sometimes challenging to collect enough training instances. Since PCC learns from developer specific changes, it may be more challenging to collect enough training instances for each developer. Therefore, it is important to understand the relationship between the number of instances in a training set and the classification performance.

For this experiment, we use the same data sets and the same parameters as Section 7.1. For each iteration in 10-fold cross validation, we keep the same test set, but we select different numbers of instances from the training set. For example, we take one fold as a test set. Instead of taking the other nine folds as the training set, we gradually reduce the size of the training set. Using the reduced training sets, we build CC and PCC classifiers. For each project, we use an increment of 100 from 100 to 900 training instances. Then we test CC and PCC on the same test set.



(a) The Linux kernel (b) Xorg (c) PostgreSQL

(d) Eclipse (e) Lucene (f) Jackrabbit

Figure 7.4: NofB20 versus number of training changes for all test subjects. X-axis is the number of training changes for CC, and 1/10 of it is the number of training changes per developer for PCC.

Figure 7.2, 7.3 and 7.4 shows the relationships between the number of training instances versus F1, PofB20 and NofB20, respectively. The results show that there is an overall trend of F1, PofB20 and NofB20 increasing with the number of the training instances.

We can compare PCC and CC results by comparing the corresponding lines in Figure 7.2, 7.3 and 7.4. Although PCC outperforms CC in general, the results that show PCC and CC are very close to each other at the beginning, and then diverge. Although not shown due to space constraints, different projects generally diverge at different points before 80 training instances per developer. When there are fewer training instances, the advantage of PCC does not show, because there are not enough training instances to build an accurate classification model, while CC has enough instances since CC has 10 times more training instances than each PCC model. Overall, our data show that it is generally worthwhile to use PCC for better performance when there are 80 or more training instances per developer in the project.

Table 7.3: The effect of different classes of features on F1. M represents meta-data features. B represents bag-of-words features. C represents characteristic vector features. The values in parentheses show the deltas against the predictions using only meta-data features. For simplicity, the p-values are not shown. Instead, statistically significant deltas are bolded.

Project	Approach	F1			
		M	MB	MC	MBC
Linux	CC	0.56	0.52(−**0.04**)	0.55(−0.01)	0.54(−0.02)
	PCC	0.58	0.56(−**0.02**)	0.58(+0.00)	0.55(−**0.03**)
PostgreSQL	CC	0.62	0.61(−0.01)	0.61(−**0.01**)	0.61(−0.01)
	PCC	0.61	0.60(−**0.01**)	0.62(+0.01)	0.60(−0.01)
Xorg	CC	0.65	0.65(+0.00)	0.63(−**0.03**)	0.65(+0.00)
	PCC	0.66	0.67(+0.01)	0.66(+0.00)	0.67(+0.01)
Eclipse	CC	0.54	0.55(+0.01)	0.54(+0.00)	0.53(−0.01)
	PCC	0.57	0.59(+**0.02**)	0.58(+0.01)	0.59(+**0.02**)
Lucene	CC	0.53	0.51(−**0.02**)	0.51(−**0.02**)	0.51(−**0.02**)
	PCC	0.60	0.57(−**0.03**)	0.59(−0.01)	0.56(−**0.04**)
Jackrabbit	CC	0.71	0.72(+0.01)	0.70(−0.01)	0.72(+0.01)
	PCC	0.74	0.72(−**0.02**)	0.71(−**0.03**)	0.72(−**0.02**)
Average	CC	-	(−**0.01**)	(−0.01)	(−**0.01**)
	PCC	-	(−0.01)	(−**0.01**)	(−**0.01**)

In general, PCC outperforms CC when there are 80 or more training instances per developer.

7.4 Effect of Different Features

There are three feature classes, i.e., bag-of-words, characteristic vectors and meta-data. Except for meta-data, bag-of-words and characteristic vectors have thousands of features. If a feature class does not statistically significantly improve the performance, it is unreasonable to include this feature class. We instigate the effect of bag-of-words and characteristic vectors. We conduct

Table 7.4: The effect of different classes of features on NofB20. M represents meta-data features. B represents bag-of-words features. C represents characteristic vector features. The values in parentheses show the deltas against the predictions using only meta-data features. For simplicity, the p-values are not shown. Instead, statistically significant deltas are bolded.

Project	Approach	NofB20			
		M	MB	MC	MBC
Linux	CC	162	154(−**8**)	167(+5)	160(−2)
	PCC	189	179(−**10**)	191(+2)	179(−10)
PostgreSQL	CC	53	66(+13)	105(+**52**)	55(+2)
	PCC	208	226(+18)	211(+3)	210(+2)
Xorg	CC	111	107(−4)	109(−2)	96(−15)
	PCC	140	158(+**18**)	146(+6)	159(+**19**)
Eclipse	CC	136	99(−**37**)	128(−8)	116(−20)
	PCC	167	157(−10)	195(+**28**)	207(+**40**)
Lucene	CC	144	172(+28)	152(+8)	176(+**32**)
	PCC	256	255(−1)	265(+9)	254(−2)
Jackrabbit	CC	392	382(−10)	386(−6)	411(+19)
	PCC	456	431(−**25**)	466(+10)	449(−7)
Average	CC	-	(−2)	(+**10**)	(+7)
	PCC	-	(−3)	(+9)	(+3)

this experiment using the same settings in Section 7.1, but we use different feature combinations. Since there are only around 10 meta-data features, their cost is nearly free. We assume the presence of meta-data features.

The impact on F1 and NofB20 is shown in Table 7.3 and Table 7.4. The columns show the project name, the approach, the F1 and NofB20 with different feature combinations. The difference against the predictions using only meta-data features is shown in brackets. We apply statistical tests on the differences. The statistically significant differences are bolded. For example, the Linux kernel predicted using meta-data and bag-of-words features by CC is 0.52, which is 0.04 lower than the predictions using only meta-data. This difference is statistically significant.

Although the performance in different projects varies, the average row on the bottom surprisingly show that different feature classs have little impact on F1. All feature class combinations are 0.01 lower than the corresponding prediction using only meta-data features. Although most of these differences are in fact statistically significant, the difference is so small (0.01) that we can safely ignore it. The impact on NofB20 is very similar. Out of six comparisons, only one is

statistically significant, and the difference is much smaller than the difference between PCC and CC.

This result is very interesting and unintuitive. There are several implications. First, this result suggests that it is better to use only meta-data features with ADTree. In terms of running time, the classifiers run faster with less features. In terms of classification performance, ADTree is the best-performing classification algorithm (Section 7.3), and different feature classs have almost no impact on ADTree's performance. Bag-of-words and characteristic vector sometimes do improve the performance, but the improvement is so small that cannot justify the cost. Second, some features could confuse the classification algorithms. Intuitively, more features should yield better performance. However, our result show that it is important to understand the features before adding them. One possible research direction is feature selection.

Different feature classes have very little impact on classification performance.

Chapter 8

Discussion and Future Work

8.1 Cost Effectiveness vs. F1

As shown in our experiments, the Linux kernel and PostgreSQL have statistically significant cost effectiveness improvements while their F1 improvements are not statistically significant. We want to understand why these two metrics can yield different results. Since PCC builds one model for each developer separately, it is easier for bug patterns to stand out than with CC. PCC should yield predictions with higher confidence on average. This is confirmed by the confidence values in our experiments. The top ranked changes in PCC have higher confidence values than those of CC. Since the prediction given by the model with a higher confidence is more likely to be correct, the top ranked changes in PCC are more accurate than those of CC. In other words, PCC can have a higher cost effectiveness than CC while the difference in F1 is small.

Cost effectiveness is more relevant when there are limited resources to inspect a portion of code. Imagine a scenario that the deadline is approaching, and developers do not have enough time to inspect all predicted buggy changes. We rank changes according to the likelihood of a change being buggy. Developers can inspect the changes in the ranked order. An increase in cost effectiveness can help developers find more bugs in such a scenario. For example, PCC improves NofB20 by up to 155 compared to CC. It means that developers can discover up to 155 more bugs using the rankings provided by PCC.

On the other hand, F1 is useful in scenarios when there are enough resources to inspect all predicted buggy changes, e.g., code review. An increase in precision can reduce time that developers spend on inspecting true clean changes. An increase in recall can help developers capture more bugs. An increase in F1 suggests an increase in precision and recall. As shown by Kim et al. [31], we can sacrifice precision for recall and vice versa.

```
(dev1) '&' < −0.5:  0.815
(dev2) '&' < −2.5:  0.480
(dev3) '&' < −0.5: −0.315
```

Figure 8.1: These three ADTree nodes are from three developers' trees. The first node means that the weight of a change being buggy increases by 0.815 if developer dev1 removes any bit-wise `and` operator in this change. A change is predicted buggy if the weight is positive.

8.2 Interpretation of Defect Prediction Results

Although being useful in the build process, developers often ignore predictions that are not explained [38]. The good news is, defect prediction techniques can be improved to provide explanations, e.g., which features lead to a buggy prediction. For example, an ADTree model is a tree that shows step by step how it makes predictions based on the features [16]. Figure 8.1 shows three ADTree nodes from three developers' trees. This figure shows how the bit-wise `and` operator can affect the decision process in three developers' models. For each change, we find how many bit-wise `and` operators the change adds or removes. For example, "-1" represents removing one bit-wise `and` operator. We compare the modifications against a threshold. "<-0.5" means to check if the change removes more than 0.5 bit-wise `and` operators. In summary, if developer dev2 removes more than 2.5 bit-wise `and` operators in a change, the weight of this change being buggy increases by 0.815. We traverse the tree by visiting the node at each level that matches the change. The change is predicted buggy if the sum of the weights along the path is positive. Developers dev1 and dev2 are more likely to produce buggy changes if they remove bit-wise `and` operators. In contrast, developer dev3 is more likely to produce clean changes if she removes bit-wise `and` operators. These examples confirm that there are subtle differences between developers and the classification algorithms can catch such differences.

8.3 Threats to Validity

Subjects are all open source projects: We collect experimental data sets (Table 6.2) from only six open source projects to evaluate PCC. Therefore, these projects might not be representative of closed source projects. We do not intend to draw general conclusions about all software projects. While we believe that our personalization approach is widely applicable, its performance may vary in closed source projects and other projects that are not evaluated by this study.

Our bug data contains noise: Since we follow the traditional change classification techniques [31] to identify bug-fixing changes, our data inevitably include noise, as Bird et al. pointed out [6]. For this reason, we carefully select our subjects to have high quality commit logs. In addition, we use two projects that have manually verified bug reports, Lucene and Jackrabbit [24]. For all subjects, we manually check the noise level. We randomly sample 200 bug-fixing commits from each project and manually verify whether they are indeed bug fixes. We find that the precision and recall are reasonable: the precision and recall are 0.87 and 0.73 for the Linux kernel, and 0.86 and 0.71 for PostgreSQL [14]; the precision and recall are 0.75 and 0.64 for Xorg, 0.78 and 0.93 for Eclipse, 0.91 and 0.93 for Lucene, and 0.95 and 0.87 for Jackrabbit. These noise levels (up to 30%) should be acceptable [33]. However, this noise may affect our experimental results even though we use the same data sets for PCC, CC, and MARS. In the future, we can reduce the noise levels through advanced techniques [34, 67].

In addition, the developer information may be inaccurate for PostgreSQL. The PostgreSQL community is currently requiring that both the author field and the committer fields reflect the committer [56]. For this reason, we do not have the actual author information for many commits, which may hurt our results. In the future, we may analyze PostgreSQL's commit messages and mailing lists to extract the author information to potentially improve PCC's performance on PostgreSQL. In addition, it may be interesting to compare author-specific change classification with committer-specific change classification.

Developer and change selection might affect our results: For PCC, we select the ten most committed developers per project, and use 100 changes per developer as our data sets (Table 6.2). Note that we used the same data sets for PCC, CC and MARS. We select these top developers because there are too many inactive developers in open source projects who committed only a few changes. Removing these inactive developers from experiments is a common practice in the literature [2, 26, 42, 54]. We select the same number of changes per developer since some developers have too many changes. For example, one developer in PostgreSQL committed about 41% of the entire changes in the project history. We do not want a large number of changes from a few developers to dominate the classification results. These selections might affect our experimental results; alternative ways of selecting developers and changes remain as our future work.

Chapter 9

Conclusions

We propose personalized defect prediction, and apply it to change classification as a proof of concept. In addition, we propose PCC+ to further improve performance by combining models based on confidence measures. Our empirical evaluation of PCC on six open source projects shows that our personalized change classification outperforms the traditional change classification [31] and MARS [5]. We also find that the advantage of PCC is not bounded to any classification algorithm. In general, PCC outperforms CC when there are more than 80 training instances per developer.

Our personalized idea could be applied to recommendation systems [25, 47, 68] and other types of predictions, such as top crashes [30], quality attributes [36], bug-fixing commits [65], vulnerabilities [61], bug location and number of bugs [52]. Since each developer may have different software development behaviors, personalized systems can effectively learn from a developer and provide useful information for the developer.

References

[1] Snowball. http://snowball.tartarus.org/.

[2] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *ICSE*, pages 361–370, 2006.

[3] Erik Arisholm, Lionel C. Briand, and Magnus Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. pages 215–224, 2007.

[4] N. Attoh-okine, S. Mensah, M. Nawaiseh, and D. Hall. Using multivariate adaptive regression splines (mars) in pavement roughness prediction. *Strategy*, 2001.

[5] Nicolas Bettenburg, Meiyappan Nagappan, and Ahmed E. Hassan. Think locally, act globally: Improving defect and effort prediction models. In *MSR*, pages 60–69, 2012.

[6] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *FSE*, pages 121–130, 2009.

[7] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1981.

[8] G. Bougie, C. Treude, D.M. Germán, and M. Storey. A comparative exploration of FreeBSD bug lifetimes. In *MSR*, pages 106–109, 2010.

[9] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. MIT Computer Science and Artificial Intelligence Laboratory Abstracts, March 2004.

[10] Chih-Chung Chang and Chih-Jen Lin. LIBSVM - A library for support vector machines, 2001.

[11] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP*, pages 73–88, 2001.

[12] Toma Curk, Janez Demar, Qikai Xu, Gregor Leban, Uro Petrovi, Ivan Bratko, Gad Shaulsky, and Bla Zupan. Microarray data mining with visual programming. *Bioinformatics*, 21:396–398, February 2005.

[13] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

[14] Jon Eyolfson, Lin Tan, and Patrick Lam. Correlations between bugginess and time-based commit characteristics. *EMSE*, pages 1–31, 2013.

[15] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.

[16] Y. Freund and L. Mason. The alternating decision tree learning algorithm. In *ICML*, pages 124–133, 1999.

[17] Jerome H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1):1–67, 1991.

[18] Google Official Blog. Personalized search for everyone, 2009.

[19] Todd Graves, Alan Karr, J. Marron, and Harvey Siy. Predicting fault incidence using software change history. *TSE*, 26(7):653–661, 2000.

[20] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

[21] Ahmed Hassan. Predicting faults using the complexity of code changes. In *ICSE*, pages 78–88, 2009.

[22] Ahmed Hassan and Richard Holt. The top ten list: Dynamic fault prediction. In *ICSM*, pages 263–272, 2005.

[23] Israel Herraiz, Jesús González-Barahona, Gregorio Robles, and Daniel Germán. On the prediction of the evolution of libre software projects. In *ICSM*, pages 405–414, 2007.

[24] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *ICSE*, 2013.

[25] Abram Hindle, Earl Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *ICSE*, pages 837–847, 2012.

[26] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *FSE*, pages 111–120, 2009.

[27] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

[28] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *UAI*, pages 338–345, San Mateo, 1995. Morgan Kaufmann.

[29] Taghi M. Khoshgoftaar and Edward B. Allen. Ordering fault-prone software modules. *Software Quality Control*, 11(1):19–37, May 2003.

[30] Dongsun Kim, Xinming Wang, Sunghun Kim, Andreas Zeller, S. Cheung, and Sooyong Park. Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *TSE*, 37(3):430–447, 2011.

[31] Sunghun Kim, Jr. E. James Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *TSE*, 34:181–196, 2008.

[32] Sunghun Kim and E. James Whitehead, Jr. How long did it take to fix bugs? In *MSR*, pages 173–174, 2006.

[33] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *ICSE*, pages 481–490, New York, NY, USA, 2011.

[34] Sunghun Kim, Thomas Zimmermann, Kai Pan, and James Jr. Whitehead. Automatic identification of bug-introducing changes. In *ASE*, pages 81–90, 2006.

[35] Ron Kohavi. The power of decision tables. In *ECML*, pages 174–189. Springer, 1995.

[36] Sergiy S. Kolesnikov, Sven Apel, Norbert Siegmund, Stefan Sobernig, Christian Kästner, and Semah Senkaya. Predicting quality attributes of software product lines using software and network measures and sampling. In *VaMoS*, page 6, 2013.

[37] Niels Landwehr, Mark Hall, and Eibe Frank. Logistic model trees. 95(1-2):161–205, 2005.

[38] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E James Whitehead Jr. Does bug prediction support human developers? findings from a google case study. In *ICSE*, pages 372–381, 2013.

[39] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *ASID*, October 2006.

[40] Markus Lumpe, Rajesh Vasa, Tim Menzies, Rebecca Rush, and Burak Turhan. Learning better inspection optimization policies. *IJSEKE*, 22(5):621–644, 2012.

[41] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken-ichi Matsumoto, and Masahide Nakamura. An analysis of developer metrics for fault prediction. In *PROMISE*, pages 18:1–18:9, 2010.

[42] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *MSR*, pages 131–140, 2009.

[43] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *FSE*, pages 13–23, 2008.

[44] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayse Basar Bener. Defect prediction from static code features: Current results, limitations, new approaches. *ASE*, 2010.

[45] Audris Mockus, David Weiss, and Ping Zhang. Understanding and predicting effort in software projects. In *ICSE*, 2003.

[46] A. W. Moore. Cross-validation, 2008.

[47] Kivanç Muslu, Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Improving ide recommendations by considering global implications of existing recommendations. In *ICSE*, pages 1349–1352, 2012.

[48] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *ICSE*, pages 284–292, 2005.

[49] National Institute of Standards and Technlogy (NIST). The economic impacts of inadequate infrastructure for software testing. http://www.nist.gov/director/prog-ofc/report02-3.pdf, May 2002.

[50] K.-M. Osei-Bryson and M. Ko. Exploring the relationship between information technology investments and firm performance using regression splines analysis. *Information and Management*, 42(1):1–13, 2004.

[51] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *ISSTA*, pages 86–96, 2004.

[52] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Predicting the location and number of faults in large software systems. *TSE*, 31(4):340–355, April 2005.

[53] Thomas J Ostrand, Elaine J Weyuker, Robert M Bell, Park Avenue, and Florham Park. Programmer-based fault prediction. In *PROMISE*, pages 1–10, 2010.

[54] J. Park, M. Lee, Jinhan Kim, S. Hwang, and Sunghun Kim. CosTriage: A cost-aware triage algorithm for bug reporting systems. In *AAAI*, 2011.

[55] Daryl Posnett, Raissa DSouza, Prem Devanbu, and Vladimir Filkov. Dual ecological measures of focus in software development. In *ICSE*, 2013.

[56] PostgreSQL Community. Committing with Git – PostgreSQL Wiki, 2012.

[57] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: A fine-grained study of authorship. In *ICSE*, pages 491–500, 2011.

[58] Foyzur Rahman and Premkumar Devanbu. How, and Why, process metrics are better. In *ICSE*, 2013.

[59] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *FSE*, 2012.

[60] B. Raskutti, H.L. Ferra, and A. Kowalczyk. Second-order features for maximizing text classification performance. *ECML*, pages 419–430, 2001.

[61] Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? *EMSE*, 18(1):25–59, 2013.

[62] Shivkumar Shivaji, E. James Whitehead Jr., Ram Akella, and Sunghun Kim. Reducing features to improve code change-based bug prediction. *TSE*, 39(4):552–569, 2013.

[63] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do Changes Induce Fixes? In *MSR*, pages 24–28, 2005.

[64] Didi Surian, Yuan Tian, David Lo, Hong Cheng, and Ee-Peng Lim. Predicting project outcome leveraging socio-technical network patterns. In *CSMR*, pages 47–56, 2013.

[65] Yuan Tian, Julia Lawall, and David Lo. Identifying Linux bug fixing patches. In *ICSE*, pages 386–396, 2012.

[66] Catherine Tucker. Social Networks, Personalized Advertising, and Privacy Controls. In *WEIS*, 2011.

[67] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and S. Cheung. Relink: Recovering links between bugs and changes. In *FSE*, pages 15–25, 2011.

[68] Yunwen Ye and Gerhard Fischer. Supporting reuse by delivering task-relevant and personalized information. In *ICSE*, pages 513–523, 2002.

[69] T. P. York and L. J. Eaves. Common disease analysis using multi- variate adaptive regression splines (mars): Genetic analysis workshop 12 simulated sequence data. *Genetic Epidemiology*, 21 Suppl 1:S649–S654, 2001.

[70] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *ICSE*, pages 14–24, 2012.

[71] T. Zimmermann, R. Premraj, and Andreas Zeller. Predicting Defects for Eclipse. In *PROMISE*, 2007.

[72] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE*, 2008.

[73] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *FSE*, pages 91–100, 2009.