# Online Defect Prediction for Imbalanced Data

by

Ming Tan

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, 2015

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

Many defect prediction techniques are proposed to improve software reliability. Change classification predicts defects at the change level, where a change is a collection of the modifications to one file in a commit. In this thesis, we conduct the first study of applying change classification in practice and share the lessons we learned.

We identify two issues in the prediction process, both of which contribute to the low prediction performance. First, the data are imbalanced—there are much fewer buggy changes than clean changes. Second, the commonly used cross-validation approach is inappropriate for evaluating the performance of change classification. To address these challenges, we apply and adapt online change classification to evaluate the prediction and use resampling, updatable classification techniques as well as remove the testing-related changes to improve the classification performance.

We perform the improved change classification techniques on one proprietary and six open source projects. Our results show that resampling and updatable classification techniques improve the precision of change classification by 12.2–89.5% or 6.4–34.8 percentage points (pp.) on the seven projects. Additionally, removing testing-related changes improves F1 by 62.2–3411.1% or 19.4–61.4 pp. on the six open source projects with a comparable value of precision achieved.

Furthermore, we integrate change classification in the development process of the proprietary project. We have learned the following lessons: *1*) new solutions are needed to convince developers to use and believe prediction results, and prediction results need to be actionable, *2*) new and improved classification algorithms are needed to explain the prediction results, and insensible and unactionable explanations need to be filtered or refined, and *3*) new techniques are needed to improve the relatively low precision.

## Acknowledgements

I wish to express my deepest appreciation for my supervisor—Professor Lin Tan. When I set out on the journey into software engineering, I never guessed how much Lin would guide me along the way and how far I would travel beyond the starting point. My academic life fell under the influence of her contagious enthusiasm and continuous dedication to research. This work is accomplished under every hand-in-hand direction from Lin.

I mainly fulfilled this task when I was an intern in Cisco Systems. My sincere gratitude for Cisco lies in every spontaneous discussion and every piece of work completed with either instruction or help from Sashank Dara and Caleb Mayeux. And of course all the paperwork my manager Ashok Javvaji helped me with.

Thanks for Professor Ladan Tahvildari and Professor Mark Crowley, for being my thesis readers, spending their valuable time to read the thesis and provide valuable feedback; and thanks for my labmates: Jiuqiu Yang, Thibaud Lutellier, Song Wang, Edmund Wong, and Richa Bindra for helping me prepare the thesis and seminar step by step.

I also want to thank all my labmates and colleagues for their valuable suggestions and patient tutorials. It is their intelligence and the interaction between us that make me firm and practiced.

Additionally, every piece of my growth is recorded with my dearest writing tutor—Jane Russwurm. Her professionalism and optimism have a great influence on me: not only on my writing, but also on my attitude towards life.

With little time accompanying them, I owe an irredeemable debt to my family and friends. Thanks to my parents and my sister Lily for granting me the power to keep going; and thanks to my best friend Monkey for unconditional support.

# Dedication

To my wild and reckless youth.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Software bugs exist all along the process of software development and may cause severe consequences. There are many ways to find bugs, such as code review, static or dynamic analysis, etc. Software defect prediction is also one of these techniques. It leverages information such as code complexity, code authors and software development history to predict code areas that potentially contain defects [5, 18, 21, 23, 31, 37, 39, 40, 64–66]. Code areas that contain defects are also referred to as *buggy* code areas. Defect prediction techniques typically predict defects in a component [9, 18, 40, 41], a file [5, 33, 37, 39], a method [17], or a change [21,31,32,47]. A *change* is the committed code in a single file [31]. Following the prior work [31], we refer to change level prediction as *change classification*.

Compared to file level prediction, the application of change level prediction has its benefits and challenges. Change classification [26, 31, 32, 47] can predict whether a change is buggy at the time of the commit, allowing developers to act on the prediction results as soon as a commit is made. In addition, since a change is typically smaller than a file, developers have much less code to examine in order to identify defects. However, for the same reason, it is more difficult to predict on changes accurately.

A recent study [35] reports the experience and lessons learned of predicting buggy files at Google. To the best of our knowledge, there are no published case studies of the application of change classification on a proprietary code base in industry. In this thesis, we apply change classification on a proprietary code base at Cisco and share the experience and lessons learned.

Achieving high prediction precision is important for the adoption of change classification in practice. If a developer finds that many predicted buggy changes contain no real bugs, i.e., the changes are *clean*, developers are likely to consider this tool as useless and thus ignore all prediction results. We apply change classification techniques [26] on one of Cisco's code bases and adapt them for integration in practice. We find that the precision is only 18.5%, which is significantly lower than the reported precisions on open source projects [26, 31]. We have identified the following main reasons.

Irrespective of proprietary or open source, the data sets are *imbalanced*. In other words, there are much more clean changes than buggy changes. The six evaluated open source projects, Linux kernel, PostgreSQL, Xorg, Eclipse, Lucene, and Jackrabbit, have buggy rates of 14.7–37.4%; and the buggy rate of the proprietary project is below 14% (Table 4.2 in Chapter 4). Classifying imbalanced data is an open challenge in the machine learning community [13, 45, 56]. Thus, solutions to address the imbalanced data challenge should improve the performance, e.g., precision and recall (Section 4.1), of change classification on both proprietary and open source projects.

K-fold cross-validation is commonly used to evaluate software defect prediction [5,18,26, 31,32,66]. Specifically, $k$-fold cross-validation randomly divides a data set into $k$ partitions ($k > 1$) and uses $(k - 1)$ partitions to train the prediction model and the remaining 1 partition as the test set to evaluate the model.

However, cross-validation is inappropriate for estimating the performance of change classification because the data points, i.e., changes, follow a certain order in time. Randomly partitioning the data set may cause a model to use future knowledge that should not be known at the time of prediction to predict changes in the past. For example, cross-validation may use information regarding a change committed in 2014 to predict whether a change committed in 2012 is buggy or clean. This scenario would not be a real case in practice, because at the time of prediction, which is typically soon after the change is committed in 2012 for earlier detection of bugs, the change committed in 2014 is nonexistent yet. Using cross-validation could also cause the data to be labeled ahead by currently unknown data. Section 2.2 provides the details of these problems.

In practice, we need a new approach to correctly reflect the real prediction scenario. In other words, at time $t$ when a change $c$ is committed, the information used to classify $c$ or any change before $c$ should be information known until time $t$ only. We call this model as *time sensitive change classification*. To test the impact of cross-validation, we applied both 10-fold cross-validation and time sensitive change classification on seven projects (one proprietary and six open source). We found that the precision of time sensitive change classification is only 18.5–59.9%, while the precision of cross-validation is 55.5–72.0% for the same data (details in Chapter 4). The results suggest that cross-validation presents a false impression of higher precisions of change classification techniques.

In summary, this thesis makes the following contributions:

- We apply and adapt time sensitive change classification and online change classification, both addressing the existing improper setup of change classification with cross-validation.

- We leverage resampling techniques to address the imbalanced data challenge, apply updatable classification algorithms, and remove *testing-related* changes to improve classification performance as well. These techniques have improved the precision of time sensitive change classification by 12.2–89.5% or 6.4–34.8 percentage points (pp.) on the one proprietary project and six open source projects. If a technique improves the precision from $a$ to $b$, the technique improves the precision by $((b-a)/a)$% or $(b-a)$ pp. Removing testing-related changes improves F1 by 62.2–3411.1% or 19.4–61.4 pp. on the six open source projects while achieving a comparable precision.

- We conduct the first case study of integrating change classification in the development process of the proprietary project. In addition to the prediction results (whether a change is buggy or not), we generate and improve explanations from prediction models and present them to Cisco developers. Explanations were lacking in previous studies [35, 44], which makes it hard for developers to act on the prediction [35].

We conduct a qualitative study for the effectiveness of change classification by industrial integration and a quantitative study for the accuracy and helpfulness of explanations generated. Through the studies, we have learned three main lessons: *1*) since change classification is relatively new to developers, we need new solutions to convince developers to use and believe prediction results and make prediction results actionable; *2*) new and improved classification algorithms for explaining the prediction results are needed, and insensible and unactionable explanations need to be filtered or refined; and *3*) even with the improved approaches, the prediction precision is still relatively low, suggesting that new techniques are needed to improve the precision.

The thesis is organized as follows: Chapter 2 introduces the background of software defect prediction, the problem of existing change classification experimental setup, and the related work. Chapter 3 describes our approaches to address the issues and to improve the performance. The results and discussions are in Chapter 4. Chapter 5 concludes the thesis.

# Chapter 2

# Background

This chapter provides the background and related work of online change classification. Section 2.1 illustrates the process of software defect prediction. Then Section 2.2 demonstrates the problem of using cross-validation for evaluating change classification in practice. Section 2.3 explains the existing research work in software defect prediction.

## 2.1 Software Change Classification

Online change classification has the same process as general defect prediction. Figure 2.1 shows the process of change classification [26, 31], which consists of the following steps: *1*) labeling each change as buggy or clean to indicate whether the change contains bugs; *2*) extracting the features to represent the changes; *3*) using the features and labels to construct a prediction model; *4*) extracting the features for new changes and applying the model to predict their labels. In this section, we first describe the labeling process of the changes following previous work [12,26,52] and then the features that are used to represent the software changes for prediction are introduced.

### 2.1.1 Labeling Buggy and Clean Changes

The labeling process uses data from the Version Control Systems (VCS) and Bug Tracking Systems (BTS). We follow the same approach used in previous work [12, 26, 52]. A line that is deleted or changed by a bug-fixing change is a faulty line, i.e., a bug. A *bug-fixing* change is a change that fixes bugs. The most recent change that introduced the faulty line

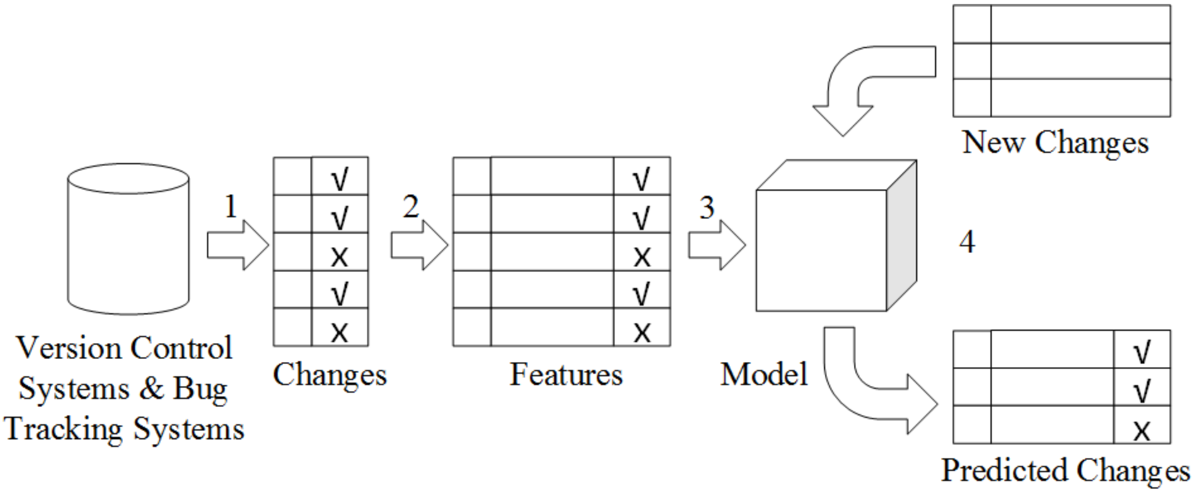

Figure 2.1: Process of Change Classification. A check mark annotates a clean change, and a cross mark annotates a buggy change.

is considered a buggy change. If a project's BTS is not well maintained and linked, we consider changes whose commit messages contain the word "fix" bug-fixing changes. If a project's BTS is well maintained and linked, we consider changes whose commit messages contain a bug report ID bug-fixing changes. Manually verified bug reports are available for Lucene and Jackrabbit [24], which are used in our study for extracting more accurate bug-fixing changes. The blaming or annotating feature of the VCS is used to find the most recent changes that modified the faulty line, which are the buggy changes. We consider the remaining changes as clean changes.

### 2.1.2 Extracting Features

Features are statistical information used to represent the changes for prediction. The types of features used in this thesis are the same as in previous work [26]: metadata, bag-of-words, and characteristic vectors.

**Metadata** Metadata describes the changes at an abstract level. It includes change size, file age, the author of a change, etc. We use all the meta-features from previous work [26], e.g., commit time and full path. In addition, we add the following features: the added line count per change, the deleted line count per change, the changed line count per change, the added chunk count per change, the deleted chunk count per change, and the changed chunk count per change. These added features aim to strengthen the existing statistical part of the prediction model.

**Bag-of-words** Bag-of-words is a commonly used feature in the machine learning community [34, 51]. The bag-of-words feature has several forms of representation. In this thesis, we use it as a vector of the count of occurrences of each word in the text. We use the snowBall stemmer [46] to group words of the same root and Weka [20] to obtain the bag-of-words features from both the commit messages and the source code.

**Characteristic Vectors** Abstract Syntax Tree (AST) is a tree based representation for a program's abstract syntactic structure. The feature of characteristic vectors counts the number of the node type in the AST representation of code. Deckard [25] is used to obtain the characteristic vector features. It first extracts the AST for each function, then uses post-order traversal to count all the nodes, finally we add the values of each node for all the functions in this file together.

Two characteristic vectors are generated: one is for the file before the change, the other is for the file after the change. We obtain a change characteristic vector by subtracting the characteristic vector before the change from the characteristic vector after the change. The change characteristic vector and the characteristic vector after the change are used as our features.

## 2.2 False High Precisions from Cross-Validation

Cross-validation is a commonly used method to evaluate the prediction models [5, 18, 26, 29, 31, 32]. The process of 10-fold cross-validation is to *1*) separate the dataset into 10 partitions randomly; *2*) use 1 partition as the test set and the other 9 partitions as the training set; *3*) repeat step 2) with a different partition as the test set until all the data have a predicted label; *4*) compute the evaluation results through comparison between the predicted labels and the actual labels of the data. This process reduces the bias in the error estimation of classification.

Using cross-validation to evaluate change classification has two problems (Figure 2.2). Changes C1–C10 are committed chronologically, where C1 is the oldest and C10 is the most recent. Dots denote buggy changes, and circles denote clean changes. An arrow links

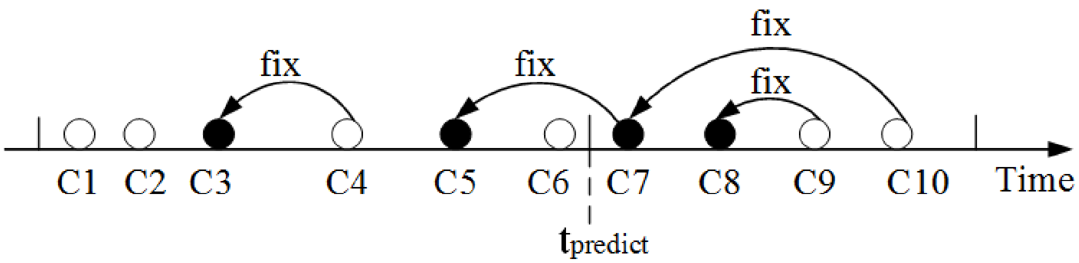

Figure 2.2: Illustrating Problems of Using Cross-Validation to Evaluate Change Classification. Dots are buggy changes, while circles are clean changes.

a buggy change and the corresponding change that fixes the buggy change. For example, C4 fixes the bug(s) in C3; therefore, C3 is a buggy change.

First, cross-validation will use future data for prediction. In this example, 10-fold cross-validation will make each change the test set in each iteration. For example, it will use C2–C10 to predict whether C1 is buggy or not, which does not match a real-world usage scenario where we typically want to make the prediction at the time when C1 is committed and by then C2–C10 are not available yet.

Second, cross-validation labels changes improperly. Using cross-validation, the labels of changes from previous work [26, 31, 32] will be as shown in Figure 2.2. For example, C5 will be labeled buggy. However, in practice, when we predict whether C6 is buggy, we would only have information at time $t_{predict}$. Therefore, C5 should be clean at time $t_{predict}$ because C7 was nonexistent at that time. It is not appropriate to consider C5 buggy when we predict the label of C6 at time $t_{predict}$, because we would not know that C5 is buggy at time $t_{predict}$.

Due to these issues, it is unclear whether precisions obtained from cross-validation are accurate estimates of precisions of defect prediction in practice. Our experiments show that cross-validation provides misleadingly high precision (Section 3.1).

## 2.3 Related Work

Researchers have contributed many work in software defect prediction and change classification. This section first discusses the time sensitive concept brought into software defect prediction at different levels. After that, the features and classifiers used by other research work are discussed. Other techniques to address the imbalanced data issue are presented next. Since we are the first one to apply change classification in practice, we also discuss other work about integrating defect prediction in industry in the end of this section.

**Time Sensitive Concept** Time sensitive and online machine learning [36] has been used to predict the evolution size of software [23]. Previous work [44] uses models built from previous software releases to predict on later releases. BugCache [33] uses the online concept to build a cache for predicting software defects. These work predict bugs at file level whereas this thesis applies with the online machine learning concept to build models to predict buggy changes. In addition, Shuo Wang et al. propose an online imbalance framework [61] first and apply it on software bug detection [60]. Note that the online learning in their work means the continuous learning process, while in this thesis online change classification is short for balanced online time sensitive change classification, which is an improved version of time sensitive change classification(Chapter 3.

**Features** In this thesis, software programs are represented by three types of features, bag-of-words, metadata and characteristic vectors. Paper [49] categorizes the representatives of the software into three types: the structural information of the software [26], the modifications made to the software [18, 26, 33], and the development information [26], e.g., dependencies, code churn, repositories etc. The three types of features cover all the three categories. There are still other work using different features. For example, Andrew Meneely et al. use the relationship between developers and files, i.e., developer network, to predict program failures [39]. Using more advanced and explainable features remain our future work.

**Classifiers** Software defect prediction uses a variety of machine learning algorithms, such as code base ensemble learning [54], Naive Bayes [26], compressed C4.5 [58], ADTree [26], neural network [2], cost-sensitive boosting neural network [63], dictionary learning [27]. Another paper [37] also uses "change classification" as their eclipse plugin tool's name. They use Support Vector Machine (SVM) as the classification algorithm to conduct file-level defect prediction when new modifications are made to the software. We use the same term "change classification" but as a different meaning. This thesis applies ADTree, resampling, and updatable classification techniques to address the specific challenges of applying change classification in practice.

**Imbalanced Data** As mentioned in Chapter 1, addressing imbalanced data issue is an open challenge. Many techniques are proposed, i.e., oversampling [57], undersampling [62], SMOTE [6], SMOTEBoost [7], negative correlation learning [59], cost-sensitive learning [63], data cleansing [19], and coding based ensemble learning [54]. We apply four of the resampling techniques to address the data imbalance issue. More novel resampling techniques remain the future work. Although previous work have applied resampling in software defect prediction at other level [7,45,57,62], to the best of our knowledge, we are the first to apply resampling and updatable classification techniques to change classification.

**General Defect Prediction in Industry** Several studies examine how well file-level defect prediction performs in practice and what developer-desired characteristics for prediction tools are [35,43,44,50]. In a short paper [4], Weyuker, Ostrand, and Bell reflect on how to measure the impact of file level prediction in practice and suggest possible approaches. This thesis applies change classification instead of file classification in practice. In addition, we provide developers explanations of the prediction results, which was lacking in these studies.

# Chapter 3

# Proposed Models and Applied Techniques for Change Classification

This chapter contains the information of our approaches to address the incorrect setup of existing change classification techniques and the imbalanced data issue. It describes time sensitive change classification (Section 3.1), followed by its improved version—online change classification (Section 3.2). Section 3.3 and 3.4 present two approaches to improve the performance of online change classification. Section 3.5 shows the approach to mitigate the negative impact brought by testing-related changes. The final section (Section 3.6) introduces the process of our case study for integrating change classification in practice.

## 3.1 Time Sensitive Change Classification

Time sensitive change classification uses information, known at time $t$ only, to classify change $c$ that is committed at $t$. For example, in Figure 2.2, time sensitive change classification predicts at time $t_{predict}$ for the change C6, i.e., the test set. The changes committed before C6 are the training set, i.e., C1-C5, which is used to build models.

However, this method has three limitations. First, in practice, we prefer to predict as soon as changes are committed so that we can examine them to identify bugs earlier. Therefore, the time period of the test set is often short, e.g., a few days or months. However, bugs typically take years to be discovered and fixed [8,11,30]. Therefore, at the prediction time $t_{predict}$, many buggy changes in the training set, especially the changes committed close to time $t_{predict}$, would not have been found and fixed yet. Therefore, many of these

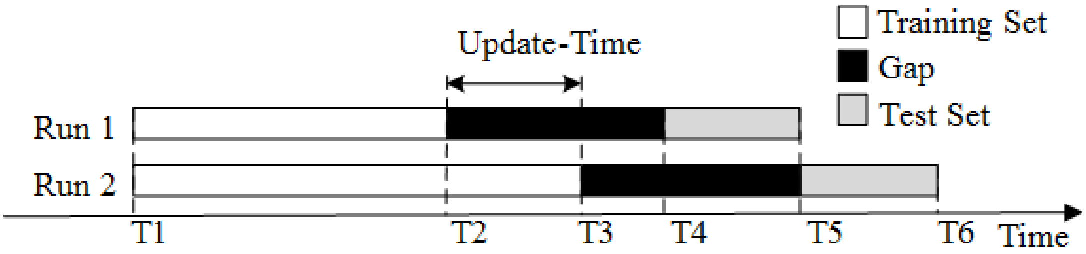

Figure 3.1: Online Change Classification.

changes in the training set, e.g., C5, will be labeled clean, indicating that the buggy rate in the training set, i.e., 1/5, will be lower than the typical buggy rate, i.e., 2/5, of a project. With fewer buggy changes, the classification algorithm may fail to learn an accurate model for the test set. In addition, the "mislabelled" clean changes become the noise data, meaning they carry the characters for buggy changes but are labeled as clean. This situation increases the noise rate for the training set, which makes it difficult for the prediction model to have a good performance.

Second, the performance of time sensitive change classification depends on the particular training set. For example, if we pick a time period that is right before a release deadline for evaluation, the changes from this time period may not be representative of the changes from other periods of time. Thus, the prediction performance of this time period may not be representative of the performance of the following normal time period.

Third, if the changes in the test set are committed over a long period of time, many development characteristics, such as the development tasks, the developer experience, and the programming styles, may be different from those of the training set. Therefore, the training set may be too old to build accurate prediction models for the test set.

## 3.2 Online Change Classification

Considering the three challenges, we developed a new approach, Balanced Online Time Sensitive Change Classification, *online change classification* in short. Online change classification is an improved version of time sensitive change classification.

To address the first challenge, we make the training set more balanced in time sensitive models. Specifically, we leave a *gap* between the training set and the test set (Figure 3.1), which allows more time for buggy changes in the training set to be discovered and fixed. For example, the time period between time $T2$ and time $T4$ is a gap. This way, the training set will be more balanced, i.e., the training set will have a higher buggy rate which is consistent with the buggy rate in the test set. A reasonable setup is to make the sum of the gap and the test set, e.g., the duration from time $T2$ to $T5$, close to typical bug-fixing time—the time from a bug is introduced until it is fixed. Our results show that using a gap of 0.2 year yields an average 14.4 pp. improvement on precision for Jackrabbit and 15.5 pp. for PostgreSQL with 1.0 year's gap.

To address the second and third challenges, we make change classification online, which applies multiple runs of time sensitive change classification. In this way, online change classification performs prediction on multiple test sets to minimize the bias from a particular test set. Correspondingly, the training set is continuously updated with new data when starting a new prediction. For each run, we add to the training set in the previous run the data immediately following the training set. The performance is the weighted average performance of these runs. Since the training set is constantly updated, the training set is more likely to have similar characteristics as the test set for constructing more accurate models.

Figure 3.1 illustrates the process of online change classification. Two runs are illustrated. The second run combines the training set in the first run and the data from time $T2$ and $T3$ (i.e., changes from $T1$ to $T3$) to form the training set for the second run. In this thesis, for simplicity, the duration from $T2$ to $T3$ (the time period to update the training set) is the same as the duration from $T4$ to $T5$, which is the duration of each test set. However, this duration can be other values for generality. The gap for the second run is $T3$ to $T5$. More recent changes (between $T5$ and $T6$) form the test set for the second run. The new prediction time is $T6$; thus, changes in the new training set will be labeled using information available at time $T6$.

## 3.3 Resampling

As discussed in Chapter 1, data for change classification are typically imbalanced, i.e., the buggy changes are much fewer than the clean changes. The clean changes are referred to as the *majority* class and the buggy changes are called the *minority* class. For software with a lower buggy rate, it is harder for classification algorithms to learn the patterns of the buggy changes.

We apply two established approaches to improve the prediction performance on imbalanced data—resampling techniques [6] and updatable classification techniques [1, 3, 10, 16, 28, 38, 48]. This section describes resampling techniques; and Section 3.4 describes the updatable classification techniques.

Resampling is an effective way to mitigate the effects of imbalanced data in change classification [45, 62]. Different algorithms are used to change the distribution between the majority class and the minority class. Two main categories of resampling techniques are oversampling and undersampling. Oversampling creates more buggy changes; whereas undersampling eliminates clean changes.

We use four types of resampling techniques to predict for the imbalanced data: simple duplicate, SMOTE, spread subsample, and resampling with/without replacement [20]. Simple duplicate and SMOTE are oversampling techniques, while spread subsample is an undersampling technique. Resampling with/without replacement can achieve both. We tune the parameters as illustrated in Section 4.1 and reserve the parameters that contribute to the highest precision.

**Simple Duplicate** randomly copies instances in the minority class to make the two classes have the same number of instances. Due to the randomness, we run this method five times and calculate the average performance.

**SMOTE** first selects instances from the minority class and finds $k$ nearest neighbors for each instance, where $k$ is a given number. It then creates new instances using the selected instances and their neighbors.

**Spread Subsample** eliminates instances in the majority class until the ratio of majority instances over minority instances is equal to a given ratio. Weight is a property of the instance that reflects the level of impact this instance has on its class. Spread subsample also updates the weight of each instance to maintain the overall weight of the two classes.

**Resampling with/without Replacement** randomly picks instances for either eliminating or duplicating until the buggy rate reaches a given value. The instances may be used multiple times for resampling with replacement whereas for resampling without replacement the instances are used only once.

## 3.4 Updatable Classification

Updatable classification algorithms update the training set incrementally to take advantage of the feedback from each run. Our online change classification updates the training set with the test set but does not benefit from the feedback from the learning process. Therefore, we apply the updatable models benefit from the feedback in each run.

We experiment with the following common updatable learning algorithms: Bayes [28], IBK [1], KStar [10], LogitBoost [16], LWL [3,14], NNge [38], and SPegasos [48]. We select these algorithms them because they are based on various basic types of machine learning algorithms, including Naive Bayes, instance-based learning, SVM, and boosting.

Bayes and LWL are based on Naive Bayes—probability learning techniques. IBK, KStar, and NNge are generally based on instance learning: they find the nearest neighbors for each instance, and all the instances are stored in the learning space in the first place. SPegasos is the algorithm that reflects the learning space into a high-dimensional space, which belongs to SVM category. LogitBoost combines logistic regression and boosting categories, which is a type of boosting learning—learning and correcting the prediction model at the same time.

## 3.5 Non Testing-related Change Classification

For the evaluated projects, the source code files contain testing code—the code that sets up and executes test cases—as well. While it is important to find bugs in testing code, developers often want to focus on predicting bugs in non-testing code, because testing code is often not shipped with software releases, thus not affecting customers.

In addition, we find that testing code is often modified due to functionality changes instead of bug fixes. Thus, if we label the testing-related changes as buggy, these reported buggy changes may be noises and could negatively affect the performance of change classification. Therefore, mitigating the negative impact from these testing-related changes is a potential way to improve the performance of change classification. We conduct a new set of experiments by removing all testing-related changes and applying online change classification on the obtained datasets. To identify testing-related changes, we use a simple heuristic that searches for changes whose full path contain the word “test”. The results are discussed in Section 4.2.

## 3.6 A Case Study of Change Classification Industrial Integration

We deploy online change classification in the software development process of the evaluated project at Cisco. The goal is to understand how to generate and present explanations of the prediction results to developers and how to improve the performance of online change classification for its adoption in software development. To achieve this goal better, we conduct a *qualitative* study in the case study. A quantitative one is conducted on historical data in Section 4.2.

In addition to predicting on each change as it is committed, it is also beneficial to predict on the changes committed in recent years since bugs typically take years to be discovered and fixed [11,30]. The recently committed changes may be buggy but have not been detected and fixed yet. Therefore, we apply our prediction models on the changes of the proprietary software made in the last year.

Before a developer makes a commit, the developer sends the `diffs` to Review Board, i.e., a new review request is created in Review Board. Review Board is the platform for peer reviews before developers actually commit to the code base. The status of a review request is open, submitted, or discarded. An open review request means the commit is not committed to the repository yet and is still in the review stage; a submitted review request means this request has passed the review stage and the change has been committed to the repository; and a discarded review request is a request that has been abandoned after code review. Discarded reviews are not in the software repository; therefore, we do not use them in our experiments. We predict on open review requests to discover buggy changes before their code review has been completed and predict on submitted review requests to discover committed buggy changes.

Change classification is integrated after a developer submits a *review request* of a commit to Review Board (Figure 3.2). Then our change classification tool automatically obtains the `diff` file—*Patches* in Figure 3.2—of this commit and extracts all the features for this commit. Next, it builds a model from this developer's recent changes. This model predicts labels of the changes in the commit. If a change is predicted buggy, our tool automatically generates an explanation of why this change is predicted buggy based on the model; then, our tool pushes the explainable results to Review Board. Review Board notifies the developer of the results. After examining the suggestions provided by change classification, the developer submits the *feedback* as whether the suggestion is taken or rejected as well as the corresponding reason.

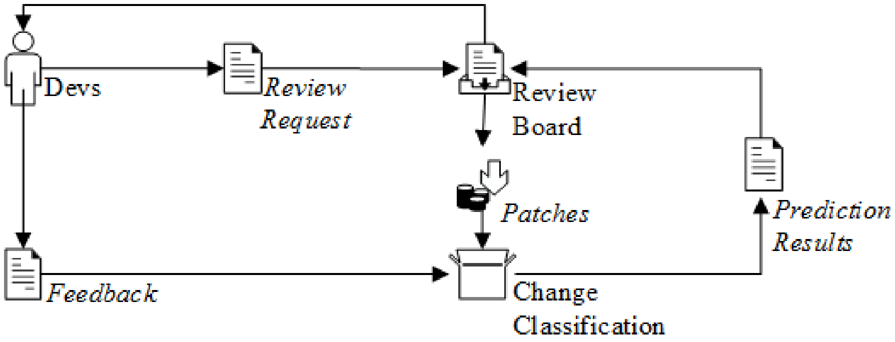

Figure 3.2: The Integration of Change Classification in the Code Review Process.

To increase the possibility of a positive adoption experience, we select developers on whose changes our tool can predict accurately. Specifically, we first build independent models for each developer who has made at least 100 buggy changes in this project. Among them, we select seven developers whose model(s) built by either resampling techniques or updatable classification could achieve 100% precision on the test sets (Table 4.7 in Chapter 4) for this case study. The test sets contain older changes that have been fixed so that we can compare the predicted labels with the actual labels of changes to measure precision. As a small case study, we start with 30 most recent review requests of each selected developer. If a developer has made fewer than 30 review requests, all of the developer's review requests are selected for this case study. Totally we have 1674 changes of 319 review requests for the field trial.

Since it is easier for developers to examine relatively small changes, we only reported the changes with 50 or fewer lines, to the developers. Totally we have predicted 96 small buggy changes of 39 review requests from the seven developers for the field trial. In our pilot study, we create four review requests regarding 11 changes from one developer and four reviewers.

The above process was considered intrusive in a production environment when integrated with Review Board. Later we opt for a less intrusive option: emailing prediction results to the developers who are interested in trying out change classification. We have emailed the prediction results of 85 changes to the remaining six developers. We also have enlisted the predictions in a file and verified their validity with a few developers in an offline process. Due to the relatively small scale of the case study, we focus on discussing our qualitative instead of quantitative results. We share the experience learned in conducting this case study in Chapter 4.3.

# Chapter 4

# Evaluation

This chapter provides the steps of setting up the experiments, tuning the parameters, and the results from all the experiments, the lessons we learned from the case study. In the end of this chapter, discussion of the limitations is presented.

## 4.1 Experimental Setup

This section first introduces the overall information of the seven projects we evaluated on. Then the process for selecting the time parameters for online change classification and tuning the parameters for the classification algorithms is explained. Besides, the metrics for the evaluation of online change classification are also discussed.

### 4.1.1 Evaluated Projects

We evaluate the change classification techniques on one proprietary project from Cisco and six open source projects, i.e., Linux, PostgreSQL, Xorg, Eclipse, Lucene, and Jackrabbit. For Xorg and Eclipse, only data from a subdirectory is used, i.e., Xserver and JDT core respectively. Table 4.1 show the overall information of these evaluated projects. The features retrieved are from the source code files[1] only.

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

Table 4.1: Evaluated Projects.

| Project | Language | LOC | First Date | Last Date | # of Changes |
|---|---|---|---|---|---|
| Proprietary | C | >10M | 2003-xx-xx | 2014-xx-xx | >100K |
| Linux | C | 7.3M | 2005-04-16 | 2010-11-21 | 429K |
| PostgreSQL | C | 289K | 1996-07-09 | 2011-01-25 | 89K |
| Xorg | C | 1.1M | 1999-11-19 | 2012-06-28 | 46K |
| Eclipse | Java | 1.5M | 2001-06-05 | 2012-07-24 | 73K |
| Lucene | Java | 828K | 2010-03-17 | 2013-01-16 | 76K |
| Jackrabbit | Java | 589K | 2004-09-13 | 2013-01-14 | 61K |

Note: Language is the programming language used for the project. LOC is lines of code. First Date is the date of the first commit of a project, while Last Date is the date of the latest commit. # of Changes is the total number of changes for each project through listed history.

### 4.1.2 Data Selection

We select changes in the middle of the software history of a project because the characteristics of a project may be unstable at the beginning of its history [22,31]. The most recent changes are excluded because buggy changes in them would not have been found and fixed yet. Note that this is for evaluation only. In practice, we predict on the latest changes so that developers can find bugs in them earlier, which is the experiment that we conducted in our case study (Section 3.6).

The specific time parameters used in online change classification are *Start-Gap*, *Gap*, *End-Gap*, and *Update-Time*.

1. *Start-Gap* is the time period at the beginning of a project where changes are excluded from our experiments.

2. *Gap*, e.g., $T2$–$T4$ or $T3$–$T5$ in Figure 3.1, is the time period between the training set and the test set. Adding such a gap allows enough time for buggy changes in the training set to be discovered by the prediction time (Section 3.2).

3. *End-Gap* is the time period at the end of a project's data collection when changes are excluded from our experiments. Typically it is the average time that it takes to discover and fix a bug, i.e., the average bug-fixing time.

4. *Update-Time*, e.g., $T2$–$T3$, in Figure 3.1, is the time period used to update the training set in the previous run to build a new model for this run.

Table 4.2: Parameters Settings for the Evaluated Projects.

| Project | SG | Gap | EG | UT | # of ExpCh | ExpBR | TrSize | TSize | NR |
|---|---|---|---|---|---|---|---|---|---|
| Proprietary | 5.0 | x.xx | x.x | 365 | >20,000 | <14.0% | >5,000 | >10,000 | 3 |
| Linux | 3.0 | 0.03 | 2.3 | 8 | 10,443 | 22.8% | 1,608 | 6,864 | 4 |
| PostgreSQL | 7.0 | 0.20 | 5.9 | 60 | 10,810 | 27.4% | 1,232 | 6,824 | 7 |
| Xorg | 5.2 | 0.40 | 5.0 | 100 | 10,956 | 14.7% | 1,756 | 6,710 | 6 |
| Eclipse | 5.0 | 0.10 | 6.2 | 40 | 9,190 | 20.5% | 1,367 | 6,974 | 6 |
| Lucene | 0.5 | 0.10 | 1.5 | 30 | 11,106 | 23.6% | 1,194 | 9,333 | 8 |
| Jackrabbit | 3.0 | 0.20 | 3.3 | 60 | 13,069 | 37.4% | 1,118 | 8,887 | 10 |

Note: SG is the Start-Gap. EG is the End-Gap. UT is Update-Time. The unit for the Start-Gap, Gap, and End-Gap is *year*, and the unit for the Update-Time is *day*. # of ExpCh is the number of changes used for the experiment, i.e., all changes excluding the changes during Start-Gap and End-Gap. ExpBR is the ratio of the buggy changes in the experimental changes. TrSize is the average size of training sets in all runs, while TSize is the total number of test instances in all runs combined. NR is the number of runs for each project.

Different time parameters are used for different projects due to their data variance, e.g., length of development history and average bug-fixing time. Initially, we follow the previous paper [26] for Start-Gaps, which are three years for most projects. To ensure enough runs of experiments, the average test set size is smaller than one fourth of the total count of experimental changes in each project. In addition, we set the quotient of the total experimental change count of each project divided by the number of runs as the upper bound of average training set size. Gap is the difference between average bug-fixing time and test set time period for each project. Update-Time is the same as the test set duration. Given the constraints, our tool automatically determines the number of runs for each project based on the data's suitability, i.e., the buggy rate in both the training set and the test set for a run should be above 0%. End-Gap is set depending on the number of runs. If the data sets cannot be determined by one run of the tool, i.e., the parameters or the data sets generated do not satisfy the constraints, the tool will add 1 month to the Start-Gap and try to find the desired data sets again.

The specific time parameters for the projects are shown in columns "SG" to "UT" of Table 4.2. The remaining columns show the number of changes in our data sets. Certain information of the proprietary project is kept vague for confidentiality. Column "# of ExpCh" is the number of changes used for the experiment, i.e., all changes excluding the changes during Start-Gap and End-Gap. All these experimental changes are used for 10-fold cross-validation. Column "ExpBR" is the ratio of the buggy changes in the experimental changes. For our online change classification, we collect data for the multiple runs from the experimental changes as described above. “TrSize” is the average size of training sets in all runs while“TSize” is the total number of test instances in all runs combined. “NR” is the number of runs for each project.

We conduct an additional experiment on selected top developers because the overall precision of online change classification is low. We want to show that we can achieve higher precision on top developers, following previous work [26]: building individual models for each developer can help improve the performance of change classification.

We rank developers by the total number of bugs in their commit history. We pick the top 10 of the developers on the list; then we select the developers whose changes allow for at least two runs and the precision of the first run is higher than 60%.

### 4.1.3 Classification Algorithms and Experiments

We use alternating decision tree (ADTree [15]) in Weka [20] as the classification algorithm since it performs best in previous work [26]. In a decision tree, one leaf shows the rule of the decision and one branch represents one decision. The path from the root to one leaf represents the learning process of the learning algorithm. One instance can satisfy multiple paths, and the final classification decision is made on the sum of all the nodes in all the satisfied paths. Our basic experiment setup uses ADTree for its simplicity, easy implementation, and good performance from previous work [26, 65, 66]. ADTree has two parameters: one is the maximum iteration time, and the other determines the number of paths to search for building the tree. We tune the maximum iteration time of ADTree the same way as previous work [26] and finally we set 10 as the maximum iteration time. We set the default value, i.e., -3, for the second parameter to search all possible paths to find the best model so that the model has a bigger chance to have a better performance.

After we obtain all the preliminary results from the above setting, we apply the resampling and the updatable classification techniques (Chapter 3). We use Weka [20] for all the resampling and updatable classification techniques except the simple duplicate method because Weka does not have an implementation of it.

We experiment with following parameter values of the classification approaches. We report the highest precision for each project, as high precision is crucial for adoption. The number of nearest neighbors in SMOTE ranges from 2 to 10. The percentage of the minority class duplications is within the range (50, 80, 100, 200, 300, 400, 500). For spread subsample, we select the best among all the ratio between the two classes varying from 1:1–10:1. As such, whether to adjust the total weight of the classes depends on the results. For the resampling with/without replacement method, all the combinations of with/without replacement and original/uniform distribution of input data are used. The output percentage of the sample size is in the same range as the SMOTE range list. For resampling without replacement, since Weka's implementation only allows undersampling, only 80 is used as the output percentage.

The tuning process for choosing the updatable classification algorithm we use is to randomly sample 100 instances from the training set with similar buggy rate (within 1%) of the original dataset. Then we run 10-fold cross-validation on the sampled data and choose the parameters that have the best results.

After tuning, the updatable classification algorithms choose the following values for different parameters: for the Bayes, we do not use uniform input distribution. Both IBK and LWL are set to use the linear nearest neighbor search algorithm; and IBK chooses one nearest neighbor. The global blending percentage in the KStar is set to 20%. The default settings (10 iterations, no internal cross-validation, and reweighting for boosting) are used for LogitBoost. NNge has two parameters: the number of attempts of generalization and the number of folders for computing the mutual information. For SPegasos, the hinge loss function is used, the regularization constant is set to 0.0001, and the epochs is set to 500.

### 4.1.4 Evaluation Metrics

There are four commonly used metrics in change classification to measure the results [40, 49]. They are *1*) *accuracy*, the percentage of correctly predicted changes in all the changes; *2*) *precision*, the percentage of correctly predicted buggy changes in all the changes which are predicted buggy; *3*) *recall*, the percentage of correctly predicted buggy changes in all the changes which are real buggy; and *4*) *F1-score*, F1 in short, the harmonic mean of the precision and recall.

However, accuracy is not suitable for evaluation when data are imbalanced. For example, when the buggy rate is 10% for a test set, if the model predicts all the changes in the test set as clean, it still achieves an accuracy of 90%. The value cannot reflect the actual ability of the model to discover the buggy changes. Therefore, we only report precision, recall, and F1 as the evaluation metrics.

## 4.2 Results

This section presents the experimental results. It is a real challenge in the research area that the prediction result can have both a high precision value and high recall value. We focus on discussing the improvement on precision because a high prediction precision is crucial for the adoption of change classification in practice as explained in Chapter 1.

We answer the following research questions (RQ):

**RQ1: Does cross-validation produce false higher precisions?** Table 4.3 shows that the precisions of the basic online change classification are 18.5–59.9% on the seven evaluated proprietary and open source projects. The precisions using cross-validation are 55.5–72.0%, which are much higher than those of basic online change classification, the one applicable in practice. The gap on precision is 7.6–37.0 percentage points (pp.) with an average of 18.4 pp. The gap on F1 is 4.4–46.7 pp., with an average of 26.6 pp. The results show that cross-validation provides false higher precisions and F1s.

**RQ2: What is the effect of resampling and updatable classification on classification performance?** Table 4.3 shows that the precisions of online change classification with resampling techniques are 33.3–73.7%. Compared to the basic online change classification ("Baseline" in Table 4.3), resampling techniques increase the precision by 12.2–89.5%, which is 6.4–34.8 pp. (13.2 pp. on average). As described in Chapter 1, if a precision is raised from $a$ to $b$, the precision is improved by $((b-a)/a)$% or $(b-a)$ pp. The precisions of online change classification with updatable classification are 30.9–59.7%. Updatable classification improves the precision of the baseline by 8.4–67.0% which is 3.8–17.3 pp. (10.6 pp. on average) for four projects. For the other three projects, it reduces the precision by 3.4 pp. on average.

Recall that we select the highest precision among all runs with different parameters because we favor higher precision over higher recall. The trade-off between precision and recall is well understood: while one increases precision, one might sacrifice recall [31]. F1 is the balanced measure of precision and recall. If one prefers a higher F1, one can select the highest F1 among all runs instead. Therefore, we have also obtained the performance results by selecting the highest F1 of all runs for the three techniques—the basic online change classification, resampling, and updatable classification (Table 4.4). These results show that resampling increases F1 by 2.2–417.2% over the basic online change classification which is 0.5–30.5 pp., 13.9 pp. on average, for all seven projects; while updatable classification improves F1 by 21.1–370.2%, which is 4.4–27.0 pp., 11.9 pp. on average, for all seven projects.

In summary, online change classification with resampling improves the precision of the basic online change classification, while updatable classification only improves precision under certain circumstances.

Table 4.3: Results of the Evaluated Projects with the Best Precision Selected.

| Project | Model | Precision | Recall | F1 |
|---|---|---:|---:|---:|
| Proprietary | Cross-Validation | 55.5% | 12.3% | 20.1% |
| | Baseline | 18.5% | 13.6% | 15.7% |
| | Resampling | **33.3%** | 7.1% | 11.7% |
| | Updatable Classification | 30.9% | **31.1%** | **31.0%** |
| Linux | Cross-Validation | 59.0% | 49.0% | 54.0% |
| | Baseline | 38.9% | **4.0%** | **7.3%** |
| | Resampling | **73.7%** | 0.9% | 1.8% |
| | Updatable Classification | 47.9% | 3.0% | 5.6% |
| PostgreSQL | Cross-Validation | 65.0% | 58.0% | 61.0% |
| | Baseline | 57.4% | 30.9% | 40.2% |
| | Resampling | **67.3%** | 9.6% | 16.8% |
| | Updatable Classification | 50.8% | **46.7%** | **48.7%** |
| Xorg | Cross-Validation | 69.0% | 62.0% | 65.0% |
| | Baseline | 42.4% | **15.6%** | **22.8%** |
| | Resampling | 49.1% | 6.3% | 11.1% |
| | Updatable Classification | **59.7%** | 7.5% | 13.4% |
| Eclipse | Cross-Validation | 59.0% | 48.0% | 53.0% |
| | Baseline | 45.1% | **17.2%** | **24.9%** |
| | Resampling | **57.6%** | 9.1% | 15.7% |
| | Updatable Classification | 48.9% | 11.9% | 19.1% |
| Lucene | Cross-Validation | 58.0% | 46.0% | 51.0% |
| | Baseline | 46.2% | 21.7% | 29.5% |
| | Resampling | **52.6%** | 15.8% | 24.3% |
| | Updatable Classification | 43.9% | **31.5%** | **36.7%** |
| Jackrabbit | Cross-Validation | 72.0% | 72.0% | 72.0% |
| | Baseline | 59.9% | 41.7% | 49.2% |
| | Resampling | **67.2%** | 21.3% | 32.3% |
| | Updatable Classification | 58.6% | **58.7%** | **58.6%** |

Note: The highest metric value among the three flavours of online change classification are bolded. The results are chosen from different algorithms based on highest precision.

**RQ3: Do testing-related changes have a negative impact on the performance of online change classification?** The results are presented in Table 4.5. From the

Table 4.4: Results of the Evaluated Projects with the Best F1 Selected.

| Project | Model | Precision | Recall | F1 |
|---|---|---:|---:|---:|
| Commercial | Baseline | 18.5% | 13.6% | 15.7% |
| | Resampling | **30.9%** | **31.1%** | **31.1%** |
| | Updatable Classification | 16.8% | 28.0% | 20.1% |
| Linux | Baseline | **38.9%** | 4.0% | 7.3% |
| | Resampling | 28.90% | 54.36% | **37.75%** |
| | Updatable Classification | 23.93% | **60.72%** | 34.33% |
| PostgreSQL | Baseline | **57.4%** | 30.9% | 40.2% |
| | Resampling | 52.7% | **68.5%** | **59.6%** |
| | Updatable Classification | 50.1% | 54.7% | 52.3% |
| Xorg | Baseline | 42.4% | 15.6% | 22.8% |
| | Resampling | **47.4%** | 15.5% | 23.3% |
| | Updatable Classification | 36.6% | **27.4%** | **34.91%** |
| Eclipse | Baseline | **45.1%** | 17.2% | 24.9% |
| | Resampling | 35.0% | **46.1%** | **39.8%** |
| | Updatable Classification | 36.6% | 33.4% | 34.9% |
| Lucene | Baseline | **46.2%** | 21.7% | 29.5% |
| | Resampling | 33.9% | **40.5%** | **36.9%** |
| | Updatable Classification | 43.9% | 31.5% | 36.7% |
| Jackrabbit | Baseline | **59.9%** | 41.7% | 49.2% |
| | Resampling | 58.6% | 58.7% | 58.6% |
| | Updatable Classification | 52.7% | **68.5%** | **59.6%** |

Note: The highest metric value among the three flavours of online change classification are bolded. The results are chosen from different algorithms based on highest F1.

results, a comparable performance on precision can be achieved on non testing-related changes. Additionally, removing testing-related files improves the recall of online change classification and leads to a significant improvement of F1. Predicting on the non-testing source code not only provides a focus for defects that developers care but also helps improve the performance of online change classification.

**RQ4: What classification performance can we achieve on more predictable developers?** Despite the improvement on precision, the overall precisions of all time sensitive change classification techniques are still low. Previous work—personalized defect

Table 4.5: Results of Non Testing-related Prediction.

| Project | Total # | T-Related Rate | T-Related | P | R | F1 |
|---|---|---|---|---|---|---|
| Linux | 10,443 | 27(0.3%) | Yes | 73.7% | 0.9% | 1.8% |
| | | | No | 73.6% | 55.4% | 63.2% |
| Postgresql | 10,810 | 157(1.5%) | Yes | 67.3% | 9.6% | 16.8% |
| | | | No | 64.4% | 37.2% | 47.2% |
| Xorg | 10,443 | 51(0.5%) | Yes | 59.7% | 7.5% | 13.4% |
| | | | No | 61.6% | 22.4% | 32.8% |
| Eclipse | 13,069 | 2,722(20.8%) | Yes | 57.6% | 9.1% | 15.7% |
| | | | No | 67.9% | 57.8% | 62.4% |
| Lucene | 8,360 | 4,009(48.0%) | Yes | 52.6% | 15.8% | 24.3% |
| | | | No | 75.2% | 78.1% | 76.7% |
| Jackrabbit | 11,106 | 3,991(35.9%) | Yes | 67.2% | 21.3% | 32.3% |
| | | | No | 54.7% | 50.4% | 52.4% |

Note: Total # means the total count of changes in the experiment. T-Related Rate means the total count of testing-related changes in the experiment. Rate means the percentage of testing-related changes in all changes. T-Related means whether the results are from the datasets that contain testing-related changes or not. Yes means datasets contain testing-related changes, otherwise not. The results listed here are selected from results generated by different classifiers with the highest precision, resampling techniques, and updatable classification techniques. P means precision, while R represents recall.

Table 4.6: Results of Top Developers' Models.

| Project | # of Devs | Precision | Recall | F1 |
|---|---|---|---|---|
| Proprietary | 5 | 75.0% | 11.8% | 20.3% |
| Linux | 2 | 85.7% | 54.5% | 66.7% |
| PostgreSQL | 3 | 73.2% | 16.5% | 26.9% |
| Xorg | 2 | 100.0% | 12.3% | 22.0% |
| Eclipse | 3 | 70.2% | 7.9% | 14.2% |
| Lucene | 3 | 78.5% | 10.1% | 17.9% |
| Jackrabbit | 4 | 78.5% | 21.8% | 34.1% |

Note: # of Devs is the count of selected developers in this project.

prediction [26]—shows that some developers may be more predictable than others, i.e., we may achieve higher prediction performance on some developers. Therefore, we select

Table 4.7: Results of the Selected Developers' Models for Case Study.

| Developer | Technique | Precision | Recall | F1 |
|---|---|---|---|---|
| 1 | Resampling | 100.0% | 100.0% | 100.0% |
| 2 | Updatable | 100.0% | 5.9% | 11.1% |
| 3 | Resampling | 100.0% | 50.0% | 66.7% |
| 4 | Resampling | 100.0% | 33.3% | 50.0% |
| 5 | Resampling | 100.0% | 50.0% | 66.7% |
| 6 | Updatable | 100.0% | 6.7% | 12.5% |
| 7 | Resampling | 100.0% | 6.7% | 12.5% |

top developers according to the standards described in Section 4.1, perform time sensitive change classification on those more predictable developers, and choose the best results for each developer. Table 4.6 shows the weighted average performance of those more predictable developers (Section 4.1). In practice, we can focus on predicting the changes from these developers to increase the chance of successful adoption of change classification in the software development process.

The result demonstrates that with better data selection, we can achieve higher precision than the basic techniques. The weighted average precisions are 70.2–100.0% for the more predictable developers from the seven evaluated projects, which are much higher than the overall precisions. This situation also indicates ensuring the quality of data is essential for the performance of change classification.

As discussed in Section 3.6, we apply online change classification techniques on the latest changes of the proprietary project in our case study. To find more predictable developers, we build prediction models for more developers in the proprietary project and select seven developers on whose changes we can achieve 100% precision (Section 3.6). The test sets contain 79–490 changes for these seven developers. Table 4.7 shows the prediction performance of the seven selected developers. Although the recalls are low, the precisions are high, which is crucial for the case study. Our results and lessons of this case study are discussed in the following section.

**RQ5: How accurate are the explanations and are they helpful for developers?** As in Section 4.3, the performance of online change classification on historical data can help researchers have a better understanding of online change classification. The fundamental assumption of the case study is that explanations generated from the classification algorithm are accurate. Here we discuss the experiments on historical data to show the statistical accuracy and helpfulness of the explanations.

We compare the explanations generated by online change classification and the corresponding bug descriptions in BTS. BTS has all the information connecting the buggy changes to the changes that fix these bugs. Online change classification collects a set of predicted buggy changes and generates the explanations. Due to the refinement of the explanations (removing all the explanations generated from non-code metrics, etc. in Section 4.3), the explanations contain only source code related features. Thus it is reasonable to consider an explanation *accurate* if the explanation contains at least one word used in the bug descriptions from BTS. Among the accurate explanations, we would like to check whether it is helpful for developers to find the root causes of bugs. We consider the explanation *helpful* if the explanation contains the words with semantic meaning in the bug root cause description.

Specifically, we generate a set of experiments from the most recent accurate data set during 2010–2012. We use the data from 2010–2011 as the training set and the data in 2012 as the test set. Online change classification is applied on these datasets and has achieved 33.8% precision, 36.5% recall, and 35.1% F1 score with the NNge algorithm (one of the updatable classification algorithms (Section 3.4)). In total, we obtain 119 changes as real buggy found by online change classification. After manual verification, we counted 90 out of 119 true buggy changes whose generated explanations are related to the bug root causes recorded in BTS; and out of the 90 buggy changes, 62 changes are easier for developers to find when they are presented with the explanations.

## 4.3 Lessons Learned

This section provides the general lessons we learned from the case study of change classification integration in software development. The lessons are either from the experience of implementation or the feedback from the developers.

**Developers need to be convinced and prediction results need to be actionable:** Compared to software testing and bug detection tools, change classification tools are newer to most developers. Therefore, one open challenge is to convince developers to use prediction results from change classification. Developers are more likely to use prediction results if they believe the results and can act on the results, i.e., the results are actionable. Through our interaction with developers, we identified three possible directions to address this challenge: *1*) presenting an explanation so that developers can understand and believe the prediction, *2*) showing the prediction precision on historical data and how it could have helped developers find the bugs they missed not having had the prediction, and *3*) integrating prediction results with test suites to prioritize test cases, e.g., upon the prediction of a buggy change, test cases related to the change are executed automatically. We have experimented with *1*) and partial explored *2*) in our case study (Section 4.2), and the experience and lessons learned are in the following two subsections. The rest of *2*), *3*), and possible combinations of them remain future work.

**Interpretable and accurate models are needed:** Both previous work [26] and our experiments show that ADTree generally outperforms other classification algorithms such as the traditional decision tree (J48) and Naive Bayes. The ADTree algorithm assigns each feature a weight and adds up the weights of all features that a change satisfies. If this sum of weights is over a threshold, a change is predicted buggy. However, the weights are typically non-integers. For example, if a change contains a modulo operator (%), then it may receive a weight of 0.13 according to an ADTree model. Developers would find such numbers confusing and unjustified. On the other hand, J48 models are more interpretable. A J48 model may show that 50 out of the 50 changes in the past that contain a modulo operator are buggy. Developers find this J48 explanation more understandable in our case study.

Since ADTree achieves a higher precision, we use the predictions from ADTree. For explanations, we obtain the X-out-of-Y numbers from ADTree models. X-out-of-Y means Y changes in the training data satisfy the above rules, and X out of them contain real bugs. Since this is not part of the ADTree implementation in Weka, we extend the implementation with this interpretation functionality. Then we use the J48 model to generate more interpretable explanations. An example of our prediction results is:

```
This change is predicted buggy with a
confidence of 100%. The possible reasons are:
> The change contains 1 or fewer "len".
> The change contains 1 or fewer "error".
> The change contains 1 or more "function-name".
> The change contains 1 or more semicolons(;).
>> 35 out of 35 changes satisfying the above
conditions contain bugs.
> The change contains 3 or fewer "char".
> The change contains 1 or more "variable-name".
```

```
>> 320 out of 407 changes satisfying the above
conditions contain bugs.
```

The function and variable names are kept anonymous for confidentiality. The bug in this change has been fixed, and this explanation points directly to the bug causes: the variable “`variable-name`” and function “`function-name`”. The developers misused the function “`function-name`”, which is a library function widely used outside the studied proprietary project. The variable suggests that the program misbehaves under the context “`variable-name`”. The variable name is specific to the target project, indicating that cross-project prediction models may have difficulty producing this explanation or predicting this buggy change, and project-specific prediction models are needed. Since the function name is generic, a cross-project prediction model may be able to identify a bug pattern across projects to predict this buggy change.

The results and experience suggest that interpretable models are crucial for the adoption of change classification. There is little work on building accurate and interpretable models for software defect prediction. Additionally, from RQ5 (Section 4.2), the tool has achieved 75.6% accuracy in generating reasonable explanations for the changes we examined and the explanations help the developers find 68.9% bugs more easily. These numbers are not good enough for adoption in industry. *New and improved techniques that are both interpretable and accurate for software defect prediction are needed.*

**Explanations need to be filtered and refined:** The classification algorithms use a statistical approach to build models. These algorithms are unaware whether a feature or an explanation from its model has a semantic meaning. For example, the explanation that “the change contains 1.5 or more modulo operators (%)” makes little sense since the number of modulo operators must be an integer. A better explanation is needed. Therefore, we change such numbers to integers in our explanation. The refined explanation becomes “the change contains 2 or more modulo operators (%)”.

In addition, many explanations are non-code metrics, which are unactionable. Assume every Friday is an internal release deadline and developers code under more pressure, thus introducing more bugs. However, the explanation that “the change was committed on Friday” does not help developers: it provides little for developers to act on the change based on this explanation. In contrast, code metric related explanations are more actionable. For example, if the explanation is “the change contains a modulo operator” allows developers to double check whether the modulo operator has been used properly.

Complex features are less useful for explanations because they may be difficult for developers who are non-conversant with the features to understand. For example, we do not present characteristic vectors to developers. We would like to explore more approaches to make complex features easier for developers to understand in the future.

**Imbalanced defect data requires new solutions to improve prediction precision and recall** Software typically has imbalanced data for defect prediction. Although we can achieve reasonable precisions for selected developers, the overall precision is still low. Therefore, we need new solutions including new algorithms and new features to improve the precision and recall of change classification.

## 4.4 Threats

**Evaluated Projects** We evaluate the change classification models on seven projects, which may not represent all software. We mitigate this threat by selecting projects of different functionalities (operating systems, servers, and desktop applications) that are developed in different programming languages (C and Java) with different development paradigms (proprietary and open source). Only one proprietary project is used for experiments. One project may not be able to present the general case of all proprietary projects.

**Labeling** Following previous work [52], the labeling process is automatically completed with the annotating or blaming function in VCS. It is known that this process can introduce noise [11, 26, 32]. The noise in the data can potentially harm the performance of the tool. Manually inspection of the process shows reasonable precision and recall on open source projects [26]. The precision and recall of the proprietary software are much higher. Previous work [32] shows that this noise level is acceptable. Existing classification techniques can tolerate a reasonable amount of noise. One way to deal with this is to more accurately label the changes, the only way available currently is to manually verify the labels. Although BTS is combined for a commercial software, some noise is introduced to the model by these collections.

**Data Selection** Our experiments use parameters and thresholds to choose a better model. Different parameter and threshold values may produce different results. We would like to study the impact of these design choices in the future.

**Case Study** We only select developers with 100% precision for the case study, assuming that data from a developer with higher precision in his/her model are more predictable. We choose the models with a high precision because too many false positives would be counter-productive for adoption. This limits the possibility for more developers to participate in the field trial. Therefore, the feedback from the developers may not be representative. Since we rule out the developers with lower precisions, there is a big chance we also exclude the developers whose data are experiment-able. For instance, data from a developer with 80% precision can also be predictable. Extending our case study to more developers and reducing the impact of confirmation bias [67] remain as our future work. In addition, because we use keyword matching in the quantitative study for the explanations, the explanations can only point to the bugs whose bug causes are explained explicitly by the existing words in the source code; in other words, it cannot infer from the information provided by the features. Further more, even though the features after refinement are all representative for only source code, we didn't get the feedback from developers. The definitions of the explanation accuracy and helpfulness may not match what developers require. A future work to get detailed developers' objective feedback is necessary.

# Chapter 5

# Conclusions and Future Work

In this chapter, we conclude the work we've done first. Then we outline and explain the future directions for this work.

## 5.1 Conclusions

In this thesis, we apply and adapt online change classification to address the incorrect evaluation presented by cross-validation, and apply the resampling techniques and updatable classification to improve the performance. Our evaluation on one proprietary and six open source projects shows that both resampling techniques and updatable classification improve the precision by 12.2–89.5% or 6.4–34.8 percentage points. Removing testing-related changes can improve F1 by 62.2–3411.1% or 19.4–61.4 pp. while achieving a comparable precision.

Our case study and experiments show that new approaches to convince developers to use prediction results are needed. In addition, interpretable prediction models are needed for software defect prediction, and new techniques are needed to improve the prediction precision for wider adoption in industry.

## 5.2 Future Research Directions

In the future, we would like to study the impact of our data selection, conduct a larger study on more developers, and propose new techniques to improve the precision of change classification, e.g., using features from BTS. Furthermore, we would like to explore other approaches of integrating change classification, e.g., for test case prioritization, risk management, quality control, process improvement, and project planning, where the prediction results are presented to various stakeholders such as developers, managers and QAs. The following paragraphs have some detailed discussions about some directions.

**Mixed Features** Features used by existing techniques include structural features (e.g., characteristic vectors), semantic features (e.g., bag-of-words), and contextual features (e.g., n-gram model [53, 55]). These features can only represent one type of the information of the software. In this thesis, although two types of the features (structural features and semantic features) are used to train the prediction model, the link between the structural information and the semantic information for the software is missing.

There are two promising ways to deal with this situation. One way is to use a type of features not only representing the structural information of the software but also the semantical meaning of the software. A recent work [42] proposes a graph-based structure to store both structural and semantic information from code. Regardless of the efficiency of this proposed model, the graph can be used as the features for better software representation. The other way is to find an approach to restoring the relationship between the structure-based features and context-based features instead of simply using them as separate features. Currently, this method does not have any existing work yet.

**Semi-supervised Learning** Recall that we exclude the data during the time period we set as Gap. However, the instances during the gap can still have valuable information for the prediction model. For example, the changes that are buggy during the gap can be included in the training set. In this way, not only is the buggy rate of the training set increased, but also more recent development information is included.

In addition, the other changes in the gap can also help better distinguish the buggy class from the clean class when they are left unlabeled. Applying semi-supervised learning may be a promising solution to leverage those unlabeled data. Semi-supervised algorithms take advantage of both labeled and unlabeled data. Thus, we can leave the data during the gap as unlabeled to utilize the useful information provided by the data during the gap.

**Test Case Prioritization** Through software defect prediction results, we could calculate the *fault density* of a file. We define fault density as the number of predicted buggy changes this file contains during the test set time period. We use the fault density as a factor that affects the fault-prone level of the source code area. The files have a ranking with their fault density. This information can be used for QA or testing team to focus on test cases that execute code areas with a high fault density. Such information may also help managers and developers allocate resources for software development and risk management.

# References

[1] David W. Aha, Dennis Kibler, and Marc K. Albert. Instance-based learning algorithms. *Mach. Learn.*, 6(1):37–66, January 1991.

[2] Mehdi Amoui, Mazeiar Salehie, and Ladan Tahvildari. Temporal software change prediction using neural networks. pages 380–385, 2007.

[3] Christopher G. Atkeson, Andrew W. Moore, and Stefan Schaal. Locally weighted learning. *Artif. Intell. Rev.*, 11(1-5):11–73, February 1997.

[4] Robert M. Bell, Elaine J. Weyuker, and Thomas J. Ostrand. Assessing the impact of using fault prediction in industry. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 561–565, March 2011.

[5] Nicolas Bettenburg, Meiyappan Nagappan, and Ahmed E. Hassan. Think locally, act globally: Improving defect and effort prediction models. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR'12, pages 60–69, Piscataway, NJ, USA, 2012. IEEE Press.

[6] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, June 2002.

[7] Nitesh V. Chawla, Aleksandar Lazarevic, Lawrence O. Hall, and Kevin W. Bowyer. Smoteboost: Improving prediction of the minority class in boosting. In Nada Lavra, Dragan Gamberger, Ljupo Todorovski, and Hendrik Blockeel, editors, *Knowledge Discovery in Databases: PKDD 2003*, volume 2838 of *Lecture Notes in Computer Science*, pages 107–119. Springer Berlin Heidelberg, 2003.

[8] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E. Hassan. An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR'14, pages 82–91, New York, NY, USA, 2014. ACM.

[9] Leslie Cheung, Roshanak Roshandel, Nenad Medvidovic, and Leana Golubchik. Early prediction of software component reliability. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE'08, pages 111–120, New York, NY, USA, 2008. ACM.

[10] John G. Cleary and Leonard E. Trigg. K*: An instance-based learner using an entropic distance measure. In *In Proceedings of the 12th International Conference on Machine Learning*, pages 108–114. Morgan Kaufmann, 1995.

[11] Jon Eyolfson, Lin Tan, and Patrick Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR'11, pages 153–162, New York, NY, USA, 2011. ACM.

[12] Jon Eyolfson, Lin Tan, and Patrick Lam. Correlations between bugginess and time-based commit characteristics. *Empirical Software Engineering*, 19(4):1009–1039, 2014.

[13] A. Fernndez, S. Garca, and F. Herrera. Addressing the classification with imbalanced data: Open problems and new challenges on class distribution. In Emilio Corchado, Marek Kurzyski, and Micha Woniak, editors, *Hybrid Artificial Intelligent Systems*, volume 6678 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin Heidelberg, 2011.

[14] Eibe Frank, Mark Hall, and Bernhard Pfahringer. Locally weighted naive bayes. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, UAI'03, pages 249–256, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[15] Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML'99, pages 124–133, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[16] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting, 1998.

[17] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C. Gall. Method-level bug prediction. In *Proceedings of the ACM-IEEE International Symposium on*

*Empirical Software Engineering and Measurement*, ESEM'12, pages 171–180, New York, NY, USA, 2012. ACM.

[18] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, July 2000.

[19] D. Gray, D. Bowes, N. Davey, Yi Sun, and B. Christianson. The misuse of the nasa metrics data program data sets for automated software defect prediction. In *Evaluation Assessment in Software Engineering (EASE 2011), 15th Annual Conference on*, pages 96–103, April 2011.

[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, November 2009.

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

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

[23] I. Herraiz, J.M. Gonzalez-Barahona, G. Robles, and D.M. German. On the prediction of the evolution of libre software projects. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 405–414, Oct 2007.

[24] Kim Herzig, Sascha Just, and Andreas Zeller. It&#039;s not a bug, it&#039;s a feature: How misclassification impacts bug prediction. In *Proceedings of the 35st International Conference on Software Engineering*, ICSE'13, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press.

[25] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE'07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[26] Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 279–289, Nov 2013.

[27] Xiao-Yuan Jing, Shi Ying, Zhi-Wu Zhang, Shan-Shan Wu, and Jin Liu. Dictionary learning based software defect prediction. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 414–423, New York, NY, USA, 2014. ACM.

[28] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, UAI'95, pages 338–345, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[29] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Trans. Softw. Eng.*, 39(6):757–773, June 2013.

[30] Sunghun Kim and E. James Whitehead, Jr. How long did it take to fix bugs? In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR'06, pages 173–174, New York, NY, USA, 2006. ACM.

[31] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, March 2008.

[32] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE'11, pages 481–490, New York, NY, USA, 2011. ACM.

[33] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE'07, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[34] Youngjoong Ko. A study of term weighting schemes using class information for text classification. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'12, pages 1029–1030, New York, NY, USA, 2012. ACM.

[35] 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 *Proceedings of the 2013 International Conference on Software Engineering*, ICSE'13, pages 372–381, Piscataway, NJ, USA, 2013. IEEE Press.

[36] Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Mach. Learn.*, 2(4):285–318, April 1988.

[37] Janaki T. Madhavan and E. James Whitehead, Jr. Predicting buggy changes inside an integrated development environment. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse'07, pages 36–40, New York, NY, USA, 2007. ACM.

[38] Brent Martin. Instance-based learning : Nearest neighbor with generalization. Technical report, 1995.

[39] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT'08/FSE-16, pages 13–23, New York, NY, USA, 2008. ACM.

[40] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engg.*, 17(4):375–407, December 2010.

[41] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 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, 2006. ACM.

[42] A.T. Nguyen and T.N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the 37st International Conference on Software Engineering*, ICSE'15, Washington, DC, USA. IEEE Computer Society.

[43] Thomas J. Ostrand and Elaine J. Weyuker. An industrial research program in software fault prediction. In *Software Engineering 2007 - Beiträge zu den Workshops, Fachtagung des GI-Fachbereichs Softwaretechnik, 27.-30.3.2007 in Hamburg*, pages 21–28, 2007.

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

[45] L. Pelayo and S. Dick. Applying novel resampling strategies to software defect prediction. In *Fuzzy Information Processing Society, 2007. NAFIPS '07. Annual Meeting of the North American*, pages 69–72, June 2007.

[46] Martin F. Porter. Snowball: A language for stemming algorithms, 2001.

[47] Lutz Prechelt and Alexander Pepper. Why software repositories are not used for defect-insertion circumstance analysis more often: A case study. *Inf. Softw. Technol.*, 56(10):1377–1389, October 2014.

[48] Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro. Pegasos: Primal estimated sub-gradient solver for svm. In *Proceedings of the 24th International Conference on Machine Learning*, ICML'07, pages 807–814, New York, NY, USA, 2007. ACM.

[49] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Trans. Softw. Eng.*, 40(6):603–616, June 2014.

[50] Emad Shihab, Ahmed E. Hassan, Bram Adams, and Zhen Ming Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE'12, pages 62:1–62:11, New York, NY, USA, 2012. ACM.

[51] Josef Sivic and Andrew Zisserman. Efficient visual search of videos cast as text retrieval. *IEEE Trans. Pattern Anal. Mach. Intell.*, 31(4):591–606, April 2009.

[52] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 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, 2005. ACM.

[53] A. Soffer. Image categorization using texture features. In *Document Analysis and Recognition, 1997., Proceedings of the Fourth International Conference on*, volume 1, pages 233–237 vol.1, Aug 1997.

[54] Zhongbin Sun, Qinbao Song, and Xiaoyan Zhu. Using coding-based ensemble learning to improve software defect prediction. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(6):1806–1817, Nov 2012.

[55] Andrija Tomovi, Predrag Janii, and Vlado Keelj. n-gram-based classification and unsupervised hierarchical clustering of genome sequences. volume 81, pages 137 – 153, 2006.

[56] Jason Van Hulse, Taghi M. Khoshgoftaar, and Amri Napolitano. Experimental perspectives on learning from imbalanced data. In *Proceedings of the 24th International Conference on Machine Learning*, ICML'07, pages 935–942, New York, NY, USA, 2007. ACM.

[57] R. Vivanco, Y. Kamei, A. Monden, K. Matsumoto, and D. Jin. Using search-based metric selection and oversampling to predict fault prone modules. In *Electrical and Computer Engineering (CCECE), 2010 23rd Canadian Conference on*, pages 1–6, May 2010.

[58] Jun Wang, Beijun Shen, and Yuting Chen. Compressed c4.5 models for software defect prediction. In *Proceedings of the 2012 12th International Conference on Quality Software*, QSIC'12, pages 13–16, Washington, DC, USA, 2012. IEEE Computer Society.

[59] Shuo Wang, Huanhuan Chen, and Xin Yao. Negative correlation learning for classification ensembles. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–8, July 2010.

[60] Shuo Wang, Leandro L. Minku, and Xin Yao. Online class imbalance learning and its applications in fault detection. *International Journal of Computational Intelligence and Applications*, 12(4), 2013.

[61] Shuo Wang, L.L. Minku, and Xin Yao. A learning framework for online class imbalance learning. In *Computational Intelligence and Ensemble Learning (CIEL), 2013 IEEE Symposium on*, pages 36–45, April 2013.

[62] Shuo Wang and Xin Yao. Using class imbalance learning for software defect prediction. *Reliability, IEEE Transactions on*, 62(2):434–443, June 2013.

[63] Jun Zheng. Cost-sensitive boosting neural networks for software defect prediction. *Expert Systems with Applications*, 37(6):4537 – 4543, 2010.

[64] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE'08, pages 531–540, New York, NY, USA, 2008. ACM.

[65] 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 *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'09, pages 91–100, New York, NY, USA, 2009. ACM.

[66] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in*

*Software Engineering*, PROMISE'07, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society.

[67] Gl alkl and AyeBaar Bener. Influence of confirmation biases of developers on software quality: an empirical study. *Software Quality Journal*, 21(2):377–416, 2013.