Thesis for Master's Degree
2020

Actionable Defect Prediction

Jiho Shin

Department of Information and Communication Engineering

Handong Global University

Actionable Defect Prediction

즉시 조치 가능한 버그 예측

Actionable Defect Prediction

Advisor: Professor Jaechang Nam

By

Jiho Shin

Department of Information and Communication Engineering

Handong Global University

A thesis submitted to faculty of the Handong Global University in partial fulfillment of the requirements for the degree of Master of Science in the Department of Information and Communication Engineering

December 25, 2020

Approved by

Signature

Professor Jaechang Nam
Thesis Advisor

Actionable Defect Prediction

Jiho Shin

Accepted in partial fulfillment of the requirements for the degree of
Master of Science

December 25, 2020

Thesis advisor Signature
Prof. Jaechang Nam

Committee Member Signature
Prof. COMMITTEE NAME2

Committee Member Signature
Prof. COMMITTEE NAME3

ICE Jiho Shin 신지호
21931006

Actionable Defect Prediction

즉시 조치 가능한 버그 예측

Department of Information and Communication Engineering,
2020, PAGE LENGTH

Advisor: Prof. Jaechang Nam

Abstract

Defect prediction studies have been actively conducted over the past decades. The existing defect prediction models face challenges in cold-start problem and the lack of actionable messages. To overcome these issues, various approaches such as Cross-Project Defect Prediction (CPDP), unsupervised defect prediction, and Just-in-time Defect Prediction (JIT-DP) have been studied. However, these studies still have limitations in several aspects. Hence, we propose a novel and potential approach of predicting defects in the commit level by searching for similar commits in existing software repositories. The model predicts a commit as "buggy" if the commit is similar to the existing Bug Inducing Commit (BIC) and if it is far away, it will predict as "clean". Then, it suggests a corresponding fix changes to the similar BIC as an actionable message for the bug-prone change. Our approach shows the potential to address the two issues of defect prediction models by alleviating the cold-start problems and suggesting patches for actionable messages.

초록 한글은 여기에 넣으세요.

CONTENTS

List of Figures

List of Tables

Chapter One

1. Introduction

As the complexity of software rises through time, the cost of quality assurance and maintenance of software development continues to arise as well. To address this issue, numerous studies have been conducted for automated quality assurance tasks to reduce the cost of software development and maintenance, such as defect prediction models. However, there are several limitations in previously proposed approaches for defect prediction

One of the limitations of defect prediction is that it lacks actionable or explainable messages [15]. The traditional defect prediction model predicts source code artifacts, usually in a file or a method level, as risky according to their degree of complexity. However, the prediction does not specify which part of the code is buggy or explains what problems the code is causing. Due to this nature, it is difficult for the developer to act upon the prediction result or understand how the module is causing the program. To alleviate this issue, Just-in-time (JIT) defect prediction got into attention. JIT defect prediction predicts bugs in the code change level. With finer granularity of prediction, researchers aim to provide 'practical' defect prediction models for developers. However, the finer granularity doesn't really solve the problem, because if the commit gets too big, the same problem occurs. In addition, the approach does still not explain why the commit is causing the problem as well. In other words, the models do not provide actionable messages for the bug-prone changes.

The second problem of traditional defect prediction models is the cold-start problem. A cold-start problem occurs when the target system lacks a history of data. Because defect prediction models are built with previous versions of the target system, it is impossible to apply on projects that are just being started. To alleviate this problem, the study of cross-project defect prediction (CPDP) and heterogeneous defect prediction (HDP) got popular. CPDP enables defect prediction to be applied to newly starting projects because the prediction models are made from other (cross-project) existing projects. HDP enables CPDP with different metrics as machine learning features. Machine learning can only be applied when the training data and the test data have the same metrics (Homogeneous), however, this case is not always met. So the development of HDP increases the range of projects that could be used as training data. However, CPDP and HDP do not fully alleviate the cold-start problem in the JIT settings because the metrics need a certain degree of historical data, i.e. the time spent after the last commit (AGE), the number of developers that contributed to the commit (NDEV), the number of unique changes in the

commit (NUC) in [8].

The last problem we want to mention is the class imbalance problem. Class imbalance problem is when a certain class of a label outweighs other classes in amount. This is very common in defect prediction scenarios because it is very hard to label modules of code as buggy. Due to the lack of label information and imbalance of class, machine learning models suffer to correctly learn and predict buggy instances.

To resolve these issues, we propose a new paradigm of defect prediction. The basic idea of our approach is to search the target change from a pool of bug inducing changes (BIC) in existing repositories and use the distance value of the search to identify the target change as buggy or clean. We call this model that searches for similar change as `SimFin` (Similar commit change Finder). `SimFin` is an auto encoder-decoder model that is learnt from a collection of buggy and clean changes in the existing repositories. By using `SimFin`, we devised `SimFinMo`, `SimFin`-based defect prediction Model. The intuition of `SimFin` is as follows. If the target change we want to predict is very similar to our searched change (with the lowest distance value, i.e. the closest change existing in a repository), it is most likely to be a buggy change. If the target change is very different from our searched change, then it is more likely to be a clean change. Our approach is very novel, because in the prediction phase, we do not apply any machine learning algorithm as opposed to other existing defect prediction models. In addition, we do not need a certain period of history data to collect metrics for test data. To predict, `SimFinMo` simply look if the distance ratio of the searched buggy and clean changes to the target change is higher than the cutoff value or not. With this approach, we could alleviate all the aforementioned limitation of defect prediction.

The contributions of our work is that:

1. We propose a completely new paradigm of defect prediction approach.
2. With our model, we can alleviate the lack of actionable messages by providing similar BFC that is associated to the searched BIC.
3. We mitigate the cold-start problem because our model is a fully universal, and needs no single previous commit for a test instance because we do not use historical metrics.
4. This approach is free from the class-imbalance problem because it does not use machine learning in the prediction phase.

Chapter Two

2. Related Work

2.1. Similar Commit Search

The key part of our approach is that SimFin finds which commits in the existing repositories that are simmilar to the target commit. Existing studies that are related to this technique are code clone detection/search, code search engines, commit clustering.

Code Clone Detection/Search

Jiang et al. [6] proposed an approach named DECARD which represents code blocks as subtrees and uses similarity algorithms on tree data structures. Lee et al. [14] proposed a method that uses multi-dimennsional indexing technique and kNN (k-Nearest Neighborhood) algorithm to reduce the search time while maitaining the functianlity of finding sematically similar code fragments. They used 54 MLOC of code to make this code clone detection module. Keivanloo et al. [9] did a similar research, but their main diference is in that they used hash tables and binary search algorithm in implementing a multi-level indexing technique. They experimented and evaluated on 266 MLOC of code bases. White et al. [30] exploited deep learning techniques that are used in natural language processing (e.g. Recursive Neural Network, or Recurrent Neural Network) to extract syntactical patterns and detect code clones with similar patterns.

The difference between code clone detection/search and similar commit search is that they have different structures of code bases. Commit shows how code is changed from one code to the other which contains information such as which nodes are added, deleted, updated, or moved, or the metadata of the commit such as which developer is responsible for the change, time of commit, number of changed files, and so on. Due to their structural differences, the necessity of studying a different approach is evident.

Code Search Engine

Bajracharya et al. [1] proposed a tool named Sourcerer which searches for code fragments. The tool divides target code fragments respect to the code usage to improve the search rate. The have divided the categories into implementation, uses, and structures. McMillan et al. [19] proposed Exemplar (Executable exmples archive) helps find code fragments that functions as the natural language query input. This study focused on improving the search rate by reducing the gap between the high abstraction of natural language query and low level language of source code. Kim et al. [10] proposed a tool

that when a user searches for a API document, it returns code snippets that can be helpful for the API's usage together with the document. Kim et al. [11] proposed FACOY (Find A Code Other than Yours) searches for a code fragment that is similar to the user's input but not in a syntactical or semantical way but with a similar function. Gu et al. [5] proposed CODEnn (Code-Description Embedding neural network) to find a semantical significance of the natural language query and the target code snippet. They do this buy mapping both natural language and code snippet in high dimennsional vector space and trains a deep learning model to map these instances as close to a space if they have semantically similar.

The difference between code search and commit search is that static code and commit has different structure, just as code clone search. Second, code search techniques are more focused on handling natural language query as input. While some do handle code fragments, they are limited in their ability to handle longer code fragments. To do a fully commit search, we must be able to handle longer code bases as input to search for commits because commits can be very long.

Commit Clustering

Kreutzer et al. [13] did a study about clustering similar commits respect to their major functions (e.g. bug fixing, refactoring, etc.). To do that, they have extracted commits that are in existing software repositories such as Git and applied LCS (Longest Common Subsequence) algorithm to retrieve a matrix of commit's similarities. With this matrix, they applied two kinds of clustering algorithms to categorize commits that have similar scores. Dias et al. [3] did a similar work but with a different scope. They categorized different changes within a commit respect to different intentions. They studied this because with a single commit, developers change several files that are sometimes nothing to do with their intentions (i.e. tangled change). To do this, they used IDE activity history, and applied different machine learning algorithms (i.e. binary logistic regression, random forest, naive bayes, etc.) for classification and applied hierarchical clustering to cluster them.

The difference between commit clustering and commit search is that in [13], the clusterings are too big to find the syntactical or semantical similarities of each commits. And as for [3], the granularity of change is within a single commit, making it hard to scale up to search similar commits in other projects.

2.2. Defect Prediction

In this section, we survey the various defect prediction methods and explain how they are different from each other and from our work.

Traditional Defect

Traditional defect prediction predicts a module in different granularity as buggy or clean. In traditional defect prediction scenario, the granularity is usually in the file-level or the method level. They use previous version of their own project to predict the current or latter version of the project. Munson et al. [21] built a classification model to classify if a module has high risk or not with the accuracy of 92%. Chidamber and Kemerer [2] proposed a suite of object-oriented related metrics that could be applied in defect prediction. Nagappan and Ball [23] proposed code churn metrics to predict defect density of the system. This was the first process related metrics and more process related metrics were proposed after.

Cross-Project Defect Prediction

Cross-project defect prediction (CPDP) was proposed to alleviate the cold-start problem of the traditional defect prediction because traditional defect prediction relied on previous versions of the target project. For project with little or no previous data, it is very hard or impossible to apply defect prediction. So CPDP uses data from other projects to learn the prediction model. Watanabe et al. [29] proposed the first CPDP approach to apply prediction model that are already built for other projects. Ma et al. [18] proposed Transfer Naive Bayes (TNB) that weights source instance similar to the target instances Nam et al. [25] proposed TCA+ to alleviate feature differencing problem in applying CPDP.

Heterogeneous Defect Prediction

Heterogeneous defect prediction was first proposed by [24]. It is a cross-project defect prediction where the source project and the target project have different feature space. This method enables source project to have different set of features which was an impossible thing to do. With this technique, it expanded the range of projects to be selected as training set, which is very important because collecting buggy data is very hard. Li et al. [16] proposed cost-sensitive transfer kernel canonical correlation analysis (CTKCCA) to evaluate nonlinear correlation relationship of the different features. Li et al. [17] proposed a two-staged ensemblme learning (TSEL) approach for HDP, which contains ensemble multi-kernel domain adaptation stage and ensemble data sampling stage. These stages handles seprates nonlinear correlation of the features and the imbalance class of the labels. Tong et al. [28] proposed a kernel spectral embedding transfer ensemble (KSETE) which addresses the class imbalance problem, finds the latent common feature space by combining kernel spectral embedding.

Just-in-Time Defect Prediction

Just-in-time defect prediction (JIT DP) tackles another problem in the traditional defect prediction. The actionability of traditional defect prediction is limited because usually a predicted module is too big, making it very hard for the developers to act upon to fix the bug. In JIT DP, the granularity of the prediction is at the change-level, usually smaller than a whole source file, making it easier for the developers to act upon due to the smaller code base. Mockus et al. [20] proposed the first to identify changes with respect to their specific reasons of causes: adding new features, correcting faults, and restructuring code for future changes. Kim et al. [12] is the first study that did a machine learning modelling for predicting buggy change of a project. Kamei et al. [7] empirically evaluated JIT prediction model in the context of cross-project scenario. They found that the models improve performance when selecting models that use other similar projects, using a larger pool of dataset, and using several projects for ensemble learning.

These various defect prediction models use machine learning for prediction. On the contrary, our method of defect prediction does not use any machine learning algorithms for prediciton. Eventhough we use autoencoder and kNN, it is for searching similar commits. The predictions are made comparing the distance and cutting them with a threshold value.

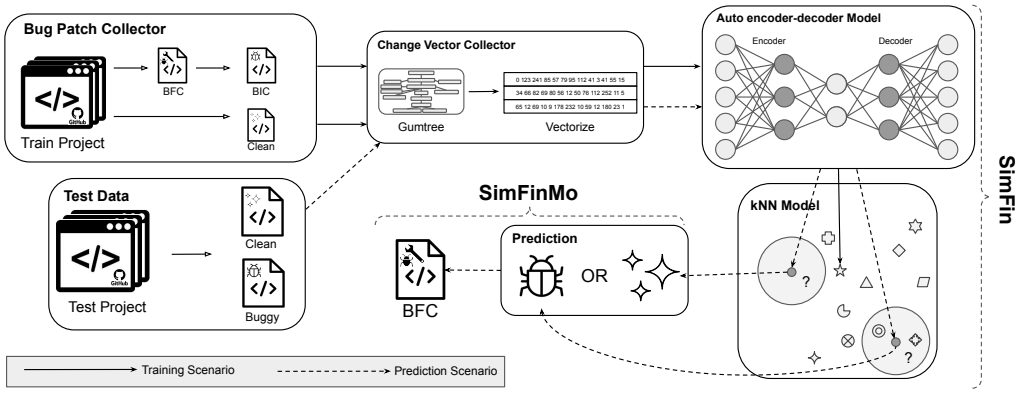

Figure 1: Overall structure of the SimFin and SimFinMo approach.

Chapter Three

3. Approach

In this section, we will explain about the details in implementing our new paradigm of defect prediction model using the similar commit search engine, SimFin. A figure of the overall approach is depicted in Fig. 1.

3.1. Collecting Bug Inducing Changes

In the beginning, we use a tool called the Bug Patch Collector to collect change data to form a training data for the SimFin. First, the tool uses SZZ algorithm [27] to collect BICs. Bug Patch Collector mines bug fixing changes (BFC) from issue tracking system such as JIRA. The issue tracking system manages all issues that have occurred during the development of the project. The issue is labeled according to the type, status and resolution. To find the bug-fix issue, we look for issues that are labelled as the following: type that are labelled “Bug”, status that are labelled “Closed” or “Resolved” and resolution that are labelled “Fixed”.

Projects that are managed by JIRA include issue keys which are unique numbers of issues and used in commit messages. Therefore, by using the issue key of bug fix issue, we can find BFCs. In BFC, it is quite probable that the deleted lines are the part that causes bugs and added lines are the corresponding patches. We compare previous commit of the BFC and the actual BFC in each source, to extract deleted or replaced lines

using git diff which is based on Myers diff algorithm [22]. Then, we apply git blame command to each modified line since git blame shows the information of last commit id, the author, the timestamp and the line number of code for the line. By using these piece of information, we can track the BIC.

After collecting the BICs, we also collect all the other changes and label it as clean changes. We do this because `SimFin` exploits both BIC and clean changes in forming the search engine.

3.2. Collecting Change Vectors

After we retrieve the change data from the Bug Patch Collector, we use the Change Vector Collector to generate vectors from the change lists, both buggy and clean. First, we collect the source code of before and after a change is applied. Then we apply the Gumtree [4] algorithm, which is a source code tree differencing algorithm with a finer granularity than that of line differencing. We use Gumtree for differencing the two source code because finer granularity captures a more precise change. By capturing a finer granularity of change, we can accurately represent change while reducing the memory as well. We also want to capture less false positive changes (code parts that are not actually changed but identified as changed) to compactly represent the changes. Lastly, because the changes are represented in AST vectors, we can capture the syntactical change in the vectors so that the `SimFin` can capture the relations of the node's syntax change.

If we apply Gumtree to the code before and after a change, the changed nodes will be represented as insertion, deletion, or update of a node or a move of a sub-tree. We only regard insertion and deletion of a node because updates and moves were mostly refactoring changes. However, they could be some other meaningful changes. So it can be regarded as sacrificing some instances for a denoising effect. After we collect each remaining changes, we encode each node-change with a unique integer value. Then, we append each value in the order of its occurrence from top to bottom, left to right in the source code. After that, we append what we call the context vector. The context vector is the a list of neighboring nodes of each node-change. We collect the context vectors by taking the all descendant nodes of the parent node of each changed node. Then we disregard the descendant nodes by collecting nodes that are within 3 lines of each corresponding changed node. We also disregard duplication as there could be a number of redundant context nodes. We have chosen to use the context vector to capture a richer information of changes, so when we search for similar changes, we not only look at the change themselves but also the context where the change have taken. The label data for `SimFin` is also constructed in this phase. For the label, we use the key value of each change, which is the commit id and the source file path of each change. The reason for making the label in such way is because when `SimFin` is given with a target commit, it

predicts the closest commit in the repository. The key value should contain the id of the commit and the source file path of the change to return the most similar commit. In JIT DP, prediction granularity is a commit level. This means if multiple files are modified in one commit, JIT DP considers this as one change. However, to avoid this large change of one commit, we consider a change in an individual file of one commit. In other words, SimFin deals with more finer changes than those in JIT DP.

3.3. SimFin: Similar Commit Change Finder

As in Fig. 1, `SimFin` is composed of a pair of deep auto encoder-decoder and k-Nearest Neighborhood (kNN) models. `SimFin` takes the vectors as Change Vector Collector returns. One set of auto encoder-decoder model and a kNN model is built from BIC instances and another set of models are built from clean instances. The change vectors are first encoded through deep feature learning using auto-encoder decoder model. Then the encoded features are fed to a kNN model to search the nearest commit change from the target commit.

Auto Encoder-Decoder Model

Auto encoder-decoder model in `SimFin` is used to learn and encode the relationship of the syntactical feature and its semantics. First, we apply zero padding to all the training instances to match dimension size. Then, the encoder encodes the vector by passing through the deep layers of the encoder network. Then it is reconstructed by passing through the deep layers of the decoder network. The reconstruction error is used to backpropagate through the network and update the weights to reduce the error. The settings we used for the auto encoder-encoder model is 10 layers for each networks, 500 nodes for each layer, 20 epochs, and a batch size of 512. We used ReLU at each layers for the activation function and a Sigmoid function at the last layer of the decoder. Binary cross-entropy was used for the loss function and Ada-delta was used for the optimizer. We use the decoder end of the networks to update the weight of the encoder model, but we only use the encoder part of the network to encode testing project in prediction phase. The BIC instances and the clean instances are trained into separate networks of auto encoder-decoder model. We trained the networks separately because we made a presumption that BIC and clean changes have different semantics.

k-Nearest Neighborhood

After we encode the syntactic and semantic representations of changes, we feed them to a kNN model to find similar changes which is the closest data point in the vector space. The kNN model originally makes prediction of an instance's class with respect to

the distance in the vector space. The labels are usually a binary or multi-class of labels, however, we use commit key as the label which is a unique label. This is done because we want to search the closest commit to the target commit. All the label in kNN model, Fig. 1, is depicted as different icons to show that each labels are unique. Because of this nature, it is not able to, or not sensible, to get the evaluation score of the kNN. Similar to auto encoder-decoder model, the kNN models are also trained separately from BIC instances and clean instances. Thus, we finally have buggy `SimFin` and clean `SimFin` respectively.

3.4. Prediction

In the prediction phase, the target commit, buggy or clean, has to go through the Change Vector Collector phase, auto encoder-decoder phase, and the kNN model phase. It is very similar to the training scenario. The change vectors, together with the context vectors, are generated from applying the Gumtree algorithm. The semantic representation of the change is learned from the encoder that is trained before-hand. Lastly, the learned feature representation is plotted in the vector space of kNN model, which then searches for nearest changes. Because we have different set of models, one (buggy `SimFin`) made from BIC and another (clean `SimFin`) from clean instances, we plot the target change in both vector spaces. After plotting the target change into both spaces, we search for the closest change in each space. Then, we can get the closest distance value in the BIC space divided by the closest distance value in the clean space. By using these two values, we can compute the distance ratio as follows:

$$DR = \frac{\delta_b}{\delta_c} \tag{1}$$

where δ_b is a the closest distance value from buggy `SimFin`while δ_c is a the closest distance value from clean `SimFin`. If a target change has a closer distance of a similar change from buggy `SimFin`than that from clean `SimFin`, DR is always less than 1. Otherwise, DR is 1 or greater. The intuition behind this method is that if a target change is very close to the closest BIC and is very far away from the closest clean instance, it is more likely to be buggy. On the contrary, if the target change is far away from the closest BIC instance but it is closer to the closest clean instance, it is more likely to be clean. Here, we need to set the cutoff value for DR values to decide whether the target change is buggy or clean. If we set a cutoff as 1 for `SimFinMo`and DR is less than this predefined cutoff value, we predict it as buggy. Since $DR = 1$ implies the target change has the same closest distance values from both boggy and clean `SimFin`, we use 1 as a default cutoff for `SimFinMo`.

After predicting a target change as buggy, then SimFinMosuggests the bug fix change (BFC) of the closest buggy change from buggy SimFin. This BFC can be used as a bug fix hint of the target change and can be an actionable message for a developer.

Chapter Four

4. Experimental Setup

This section explains about the settings of the experiment that is conducted in this study.

4.1. Research Questions

To evaluate the SimFinMo in defect prediction, we have defined two research questions.

- RQ1: Is SimFinMo comparable to various machine learners in defect prediction?
- RQ2: What are the impact of various SimFinMo cutoffs in terms of prediction performance?

From answering the research questions, we aim to investigate the effectiveness of SimFinMo by comparing the prediction performance with the existing baseline and to show different aspects of SimFinMo in using different cut-off values.

4.2. Dataset

We used 133 active, Java projects in the Apache foundation to construct SimFin. This number corresponds to the number of projects with active GitHub or JIRA issue tracking system. We have collected 133 projects and the total number of BIC collected is 44K instances and 1M of clean instances for the training data to build SimFin.

To evaluate SimFinMo, we compared it with typical JIT DP models thus we chose test data by considering JIT DP models. Table 1 shows the details of the test project. The test set used are also from Apache projects. These test projects were selected by considering various buggy ratios and the different number of commits. JIT DP conducted in a commit level still remains as very challenging to achieve high prediction performance. One of reasons is that the number of buggy commits is significantly smaller than that of clean commits. This ratios is affected by the total number of commits of a project. Thus, for our test data, we randomly choose seven projects by considering various buggy ratios and the different number of commits.

As explained in the approach section, the two SimFin models are trained by the BIC instances or clean instances. For the test set, we also use both buggy and clean instances for the prediction scenario. Since we choose the seven test projects from the 133 ASF

projects, we ignore the 'same' change found as similar by `SimFin`. By doing this, we can ignore the occurrence of the model predicting the closest commit as the ground truth.

Table 1: The list of project used as a test set

Name	# of Buggy	# of Clean	Total
jena	466 (1.1%)	43867 (98.9%)	44333
maven	988 (9.2%)	10786 (90.8%)	11774
ranger	709 (12.2%)	5810 (87.8%)	6519
sentry	265 (10.8%)	2446 (89.2%)	2711
sqoop	91 (2.2%)	4204 (97.8%)	4295
syncope	1254 (4.6%)	26415 (95.4%)	27669
tez	1091 (16.5%)	6629 (83.5%)	7720
median	709 (9.2%)	6629 (90.8%)	7720

4.3. Baseline

The baseline we use to compare the prediction performance is a typical JIT defect prediction model from Kamei et al.[8].

We used 13 out of 14 metrics of Kamei et al. [8] used. We did not use Developer experience on a subsystem(SEXP) metric because Kamei[8] builds on CVS that is centralized version control system, but projects in our experiment are based on git that distributed version control system. For this reason, we didn't consider a subsystem. The metric type is classified five dimensions that are diffusion, size, purpose, history and experience.

Diffusion dimension is figure of a distributed change. A distributed change can be measured by counting the component of source files. There are four features that are Number of modified subsystems(NS), Number of modified Directories(ND), Number of modified files(NF) and distribution of modified code(Entropy). We change subsystem of NS to package in Java. For example, There are three source files change in a commit. One is java/src/clami/main.java, another is java/src/clami/utils.java and the other is java/src/city/cat.java. Then, NS is 1 (i.e., java/src/), ND is 2 (i.e., clami/ and city/) and NF is 3 (i.e., main.java, cat.java and utils.java). Entropy counts the distribution of modified lines in the source files. **Size dimension** is number of lines in a source file. Lines of code added(LA), Lines of code deleted(LD) and Lines of code in a file before change(LT) exist. **Purpose dimension** has one feature that FIX. FIX is a label of each source file in a commit and the value is buggy or clean. **History dimension** is about the revision history of source changes from the past to the present. NDEV is the number of unique developers

who have modified a source file. AGE is the interval between the current source file time and the most recently modified time. NUC is the number of unique changes in a commit. For example there are four source files in a commit that are A, B, C and D. File A and B had been modified at α commit, file C had been modified at β commit and file D had been modified at γ commit. In this case, NUC is 3 (i.e., α, β and γ). The last dimension is **Experience dimension** that is the information of developer in the project. The developer who frequently participate in the project is less likely to cause defects because the developer understand the project well. Experience dimension has two factor : Developer experience(EXP) and Recent developer experience(REXP). EXP is the total number of commits the developer has created. REXP is the number of commits that have been weighted according to the year the developer participated in. A developer, for example, created one commit in 2017, three commit in 2018 and two commit in 2020. REXP in 2020 is 3.25 (i.e., $\frac{2}{1} + \frac{3}{3} + \frac{1}{4}$), and REXP in 2021 is 1.95 (i.e., $\frac{2}{2} + \frac{3}{4} + \frac{1}{5}$).

We use 10-fold validation to Kamei metrics for performance evaluation[]. The data set is divided into 10 sets. The 9 sets is training set that are used to make machine learning models, the other set is test set that is used to verify the performance of machine learning models.

Evaluation Metrics

The evaluation metrics for comparing `SimFinMo` and baseline are precision, recall and F-measure. We used a variety of evaluation metrics to assess a sound experiment and show various aspect of the predictors. Confusion matrix is needed to evaluation two models. There are four metrics as shown table 2 : True Positive(TP) is that the real label is true and the model predicts true. False positive(FP) is the real label is false but the model predicts true. False Negative(FN) is the real label is true but the model predicts false. True Negative(TN) is the real label is false and the model predicts true. Precision is value of positive predictive that is the correct percentage of bugs among predicted bugs ($Precision = \frac{TP}{TP+FP}$) and recall is hit rate that is the percentage of predicted bugs among actual bugs ($Recall = \frac{TP}{TP+FN}$). F-measure is the harmonic mean of the precision and recall ($F1 - measure = 2 \times \frac{Precision \times Recall}{Precision + Recall}$).

Table 2: Confusion matrix

Actual \ Predicted	Buggy	Clean
Buggy	True Positive	False Positive
Clean	False Negative	True Negative

Chapter Five

5. Experiment Results

This section shows the experimental results of the reproduced method of Kamei et al. [8] and our approach of SimFinMo. Table 3 shows the overall results of the baseline and our approach.

5.1. RQ1: Is SimFinMo comparable to various machine learners?

From Table 3, We can see that all most all of the Kamei's results has better results in precision. On the other hand, results for SimFinMo always has better performance in recall. Due to precision and recall having trade-offs with each other, it is better to see the F-measure which is the harmonic mean of precision and recall. From the results, we can see that out of 7 projects, SimFinMo outperforms 5/7 of the projects in F-measure. From the results, we can state that SimFinMo outperforms Kamei overall.

5.2. RQ2: What are the impact of various SimFinMo cutoffs in terms of prediction performance?

To provide a better concept of how well the model SimFinMo predicts defective modules, we have investigated the different performance values with different cutoff values. The results are tabulated in Table 4. Due to the limitation of space in the report, we have only tabulated one of the test projects, ranger. From the table, we can see that precision score is highest when the cut-off value converges to zero. However, the recall value is the lowest. The precision score peaks when the cut-off ranges from 0.000001 to 0.1 for other projects as well. On the other hand, precision drops pretty low when the cut-of value goes over 1. Similarly, recall continues to go up when the cut-off value gets higher. The behavior of the cut-off values are the same with machine learning. When the predictor has low cut-of value

5.3. Analysis

Table 3 shows the prediction performances between baseline and SimFinMo in various measures such as precision, recall, f-measure. The baseline based on Kamei metrics was evaluated using six classification algorithm such as Bayes Net, Naive Bayes, Random Forest, LMT, J48 and IBk.

We use Friedman and Nemenyi test to statistically evaluate the performance of algorithms of SimFinMo and baseline. Friedman test is a non-parametric test to determine

the statistical significant of the data that is classification algorithm, and usually comparing three or more data. In this paper, Friedman test is used to compare the statistical significance of evaluation metrics of all the classifiers of each project. The outputs of Friedman test are degree of freedom that is the maximum number of logically independent values, Friedman chi-squared that if the value is large, there is a relationship and if it is small, there isn't relationship and p-value that mean the relationship is statistically significant when the value is less than 0.05. The p-value of precision is 0.008772, and Friedman chi-squared is 17.143. The p-value of recall is 5.88E-06, and Friedman chi-squared is 34.303. Lastly, p-value of f-measure is 0.2179, and Friedman chi-squared is 8.2857. They all have the same degree of freedom that is 6. As a result, since the p-value of precision and recall is less than 0.05, it was statistically verified that there is a difference in performance of the algorithm. Nemenyi test has characteristics similar to Friedman test since it is usually conducted after Friedman test. It compare statistical significant between two pairwise data. By Friedman test we found that there was a difference in the defect prediction performance of the algorithm. Therefore, we conduct Nemenyi test which calculate two algorithms difference in performance. P-value is created by comparing seven classifiers with other classifiers other than oneself.

Table 3: The results of Kamei et al.[8] and `SimFinMo`

Project Name	Precision								Recall								F-measure							
	BN	NB	RF	LMT	J48	IBk	SimFinMo	Partial (1)	BN	NB	RF	LMT	J48	IBk	SimFinMo	Partial (1)	BN	NB	RF	LMT	J48	IBk	SimFinMo	Partial (1)
jena	0.080	0.025	0.685	0.500	0.647	0.190	0.024	0.016	0.436	0.603	0.107	0.034	0.024	0.178	0.446	0.612	0.135	0.047	0.186	0.064	0.046	0.184	0.046	0.031
maven	0.310	0.105	0.561	-	0.365	0.215	0.143	0.112	0.031	0.672	0.056	0.000	0.031	0.216	0.778	0.695	0.057	0.182	0.101	-	0.058	0.215	0.242	0.193
ranger	0.358	0.159	0.596	0.541	0.518	0.286	0.269	0.136	0.087	0.481	0.096	0.028	0.083	0.278	0.664	0.678	0.141	0.239	0.165	0.054	0.143	0.282	0.383	0.226
sentry	-	0.115	0.421	-	0.333	0.224	0.218	0.129	0.000	0.125	0.030	0.000	0.004	0.211	0.725	0.691	-	0.120	0.056	-	0.007	0.217	0.335	0.217
sqoop	0.104	0.053	0.800	0.286	-	0.149	0.035	0.028	0.275	0.846	0.088	0.022	0.000	0.143	0.659	0.703	0.151	0.099	0.158	0.041	-	0.146	0.066	0.055
syncope	0.478	0.097	0.538	-	-	0.210	0.124	0.059	0.009	0.058	0.061	0.000	0.000	0.193	0.681	0.680	0.017	0.073	0.110	-	-	0.201	0.210	0.108
tez	0.335	0.209	0.675	0.561	0.481	0.394	0.278	0.166	0.137	0.572	0.226	0.081	0.259	0.380	0.675	0.670	0.195	0.306	0.339	0.141	0.337	0.387	0.394	0.266
Average Rank	5.571	6.857	2.000	5.429	4.857	5.000	6.143	7.857	6.714	4.000	6.286	8.429	7.714	5.000	2.714	2.714	6.286	5.286	4.000	8.000	7.286	3.000	3.143	5.714

Table 4: This table shows different performance metrics using different cut-off values. The following result is from the project ranger.

Cutoff Value	Precision	Recall	F-measure
0.000001	0.85714286	0.00846262	0.01675978
0.1	0.79166667	0.02679831	0.05184175
0.2	0.52027027	0.10860367	0.17969662
0.3	0.38945233	0.27080395	0.31946755
0.4	0.33103448	0.40620592	0.36478784
0.5	0.294635	0.47249647	0.36294691
0.6	0.28164794	0.5303244	0.36790607
0.7	0.27102804	0.57263752	0.36792025
0.8	0.26918239	0.60366714	0.3723358
0.9	0.26666667	0.63187588	0.37505232
1	0.26868226	0.66431594	0.38261576
2	0.23634812	0.78138223	0.36292172
3	0.22235112	0.81100141	0.34901366
4	0.21674694	0.82510578	0.34330986
5	0.21528525	0.84626234	0.34324943
6	0.21344011	0.86459803	0.34236247
7	0.2122449	0.88011283	0.34201151
8	0.21022727	0.88716502	0.33990813
9	0.20771513	0.88857546	0.33671833
10	0.20715693	0.8899859	0.33608522

Chapter Six

6. Discussion

In this section, we will discuss about how this approach is important in the software engineering practice, especially in automated quality assurance technique.

6.1. Importance of this Approach

Our method of defect prediction is a novel technique which does not use machine learning algorithm in the prediction phase. In existing studies such as traditional, cross-project, heterogeneous, and just-in-time defect prediction, major issues are caused by the short comings of machine learning techniques. Cold-start problem is caused because training data is necessary for the model to be build from the target project. This issue is alleviated through the study of CPDP, however, CPDP does not perfectly solve the cold-start problem because many of the approaches use historical metrics as a feature. Because `SimFinMo` exploits existing software projects without the need of historical metrics, it fully resolves the cold-start problem.

Another major problem of defect prediction is the lack of actionable messages to act upon. Traditional defect prediction model predicted the risk of module in a file or method level. For larger projects, developers needed so much time to find the bug inside the risky module and to find the fix for it. To resolve this issue, JIT defect prediction was actively studied because a code change were usually a lot smaller than a whole file or method. By making the granularity smaller, developers can easily identify the location of the defect relatively because there are shorter amount of code to inspect. However, this does not resolve the fundamental problem. JIT defect prediction does not tell the developers how to correct the defect. By using `SimFinMo`, we can identify which change is risky and we can also show the original BFC which is associated to the BIC that has the nearest distance value. The suggested BFC might not be the exact fix for the defect, however, it can provide some ideas to assist on fixing the risky change to the developer.

6.2. Future Work

For future work, it is necessary to find the right threshold value to separate clean instances and buggy instances. Currently we used different values for different test projects which can only cut the relationship of the classes linearly. However, more studies can be proposed to distinguish this relationship in a more complex, non-linear way. Currently, it is difficult to find a global threshold that will work best for all the projects. This is crucial

```
diff --git a/src/java/org/apache/sqoop/tool/ImportTool.java b/src/java/org/apache/sqoop/tool/ImportTool.java
index ad1d48b5..ff7b822c 100644
--- a/src/java/org/apache/sqoop/tool/ImportTool.java
+++ b/src/java/org/apache/sqoop/tool/ImportTool.java
@@ -571,7 +571,7 @@ private Path getOutputPath(SqoopOptions options, String tableName, boolean temp)
  if(salt == null && options.getSqlQuery() != null) {
  salt = Integer.toHexString(options.getSqlQuery().hashCode());
  }
-  outputPath = AppendUtils.getTempAppendDir(salt);
+  outputPath = AppendUtils.getTempAppendDir(salt, options);
  LOG.debug("Using temporary folder: " + outputPath.getName());
  } else {
  // Try in this order: target-dir or warehouse-dir
```

Figure 2: Example of a ground truth BFC from project sqoop.

for the predictor to work on newly started project because there are no historical data to find the best threshold.

Also, it is promising to use the suggested BFCs as an ingredient or operation list that can be used in the automatic program repair field. In the field of automatic program repair, reducing the search space is critical to find the right patch within a feasible time. Ingredients retrieved from suggested BFC could be very helpful to reduce the search space of finding the correct patch. Fig. 2 is an instance of BFC that we collected from project sqoop, one of the test project. A BIC is retrieved from blame tracing the deleted BFC lines. When a BIC is retrieved, it searches for the most similar change in a repository. From the searched similar change, we can get the associated BFC, which is show at Fig. 3. BFC from Fig. 2 shows that a parameter of a method invocation was deleted. BFC from Fig. 3 shows that a different method was invoked with one less parameter. Even though the second change was calling a different method, we can infer that the two methods have a similar function by looking at the method name (addHtmlComponent and addComponent) and the parameters they are using. Despite the different context (different project, return type of method) we can get a hint of which operation to use in automatic program repair, which is update method, or update method parameter. With these hints, we can help automatic program repair approaches to reduce the search space of operation they use to enlarge other search spaces.

```
diff --git a/impl/src/main/java/org/apache/myfaces/view/facelets/tag/jsf/html/HtmlLibrary.java
b/impl/src/main/java/org/apache/myfaces/view/facelets/tag/jsf/html/HtmlLibrary.java
index dc5869c21..4bbda49a1 100644
--- a/impl/src/main/java/org/apache/myfaces/view/facelets/tag/jsf/html/HtmlLibrary.java
+++ b/impl/src/main/java/org/apache/myfaces/view/facelets/tag/jsf/html/HtmlLibrary.java
@@ -71,9 +71,9 @@ public final class HtmlLibrary extends AbstractHtmlLibrary

  this.addHtmlComponent("outputLink", "javax.faces.HtmlOutputLink", "javax.faces.Link");

-  this.addHtmlComponent ("outputScript", "javax.faces.Output", "javax.faces.resource.Script");
+  this.addComponent ("outputScript", "javax.faces.Output", "javax.faces.resource.Script",
HtmlOutputScriptHandler.class);

-  this.addHtmlComponent ("outputStylesheet", "javax.faces.Output", "javax.faces.resource.Stylesheet");
+  this.addComponent ("outputStylesheet", "javax.faces.Output", "javax.faces.resource.Stylesheet",
HtmlOutputStylesheetHandler.class);

  this.addHtmlComponent("outputText", "javax.faces.HtmlOutputText", "javax.faces.Text");
```

Figure 3: Example of suggested BFC from the project myfaces.

Chapter Seven

7. Threats to Validity

7.1. Construct Validity

In collecting the change vectors, we have disregarded updates and move operations from the changes due to occurrence of simple refactoring changes with no actual behavioral changes. However, this may lead in missing some real important changes such as changing the a method invocation to complete another method invocation or moving an if statement to next line to change the control flow of the program. We believe that this sacrificed some of the true positives to eliminate all the false positives. For future work, handling precise updates and move conditions will enhance the change collecting accuracy and therefore enhance the ability of `SimFinMo`.

7.2. External Validity

Although we have used 7 of Apache projects to evaluate our `SimFinMo`, it might not represent all the projects that are in software repositories. However, the projects are various in their size, domain, and development time. So we believe the results from our study has empirical value in the software engineering society.

7.3. Internal Validity

For collecting BIC data we have used the SZZ algorithm [27]. Although the algorithm is often used to collect BIC instances, it has limitations. The deleted lines from the BFC that are blamed to trace BIC are not always bug inducing as they could be refactoring or cosmetic changes [26]. However, with the circumstances, `SimFinMo` showed comparable or better performance to the existing baselines. So we believe that with the better algorithm that has better precision in BIC instance collecting will improve the performance of `SimFinMo`.

8. Conclusion

In conclusion, we have proposed a novel approach for a universal actionable defect prediction model that resolves the biggest limitation of current defect prediction model such as cold-start problem and lack of actionable messages. This study is the first of a kind that tackles to predict defective modules in change level without a machine learning

predictor. Although the suggested SimFinMo, has relatively low prediction performance we believe more studies of this direction will contribute to automated quality assurance techniques, such as defect prediction and automatic program repair. More related studies such as finding a better solution for setting the cut-off value and a way of collecting higher quality of defect and useful clean changes will enhance the prediction performance to contribute in better actionable messages.

References

Bibliography

[1] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: A search engine for open source code supporting structure-based search," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 681–682.

[2] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[3] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, 2015, pp. 341–350.

[4] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[5] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, IEEE, 2018, pp. 933–944.

[6] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE'07)*, IEEE, 2007, pp. 96–105.

[7] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2072–2106, 2016.

[8] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[9] I. Keivanloo, J. Rilling, and P. Charland, "Internet-scale real-time code clone search via multi-level indexing," in *2011 18th Working Conference on Reverse Engineering*, IEEE, 2011, pp. 23–27.

[10] J. Kim, S. Lee, S.-w. Hwang, and S. Kim, "Towards an intelligent code search engine.," in *AAAI*, 2010.

[11] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon, "Facoy: A code-to-code search engine," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 946–957.

[12] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[13] P. Kreutzer, G. Dotzler, M. Ring, B. M. Eskofier, and M. Philippsen, "Automatic clustering of code changes," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, IEEE, 2016, pp. 61–72.

[14] M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim, "Instant code clone search," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 167–176.

[15] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead, "Does bug prediction support human developers? findings from a google case study," in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 372–381.

[16] Z. Li, X.-Y. Jing, F. Wu, X. Zhu, B. Xu, and S. Ying, "Cost-sensitive transfer kernel canonical correlation analysis for heterogeneous defect prediction," *Automated Software Engineering*, vol. 25, no. 2, pp. 201–245, 2018.

[17] Z. Li, X.-Y. Jing, X. Zhu, H. Zhang, B. Xu, and S. Ying, "Heterogeneous defect prediction with two-stage ensemble learning," in *Automated Software Engineering*, vol. 26, 2019, pp. 599–651, ISBN: 1573-7535.

[18] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Information and Software Technology*, vol. 54, no. 3, pp. 248–256, 2012.

[19] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1069–1087, 2011.

[20] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases.," in *icsm*, 2000, pp. 120–130.

[21] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Transactions on software Engineering*, vol. 18, no. 5, p. 423, 1992.

[22] E. W. Myers, "Ano (nd) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1-4, pp. 251–266, 1986.

[23] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 284–292.

[24] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 9, pp. 874–896, 2017.

[25] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *2013 35th international conference on software engineering (ICSE)*, IEEE, 2013, pp. 382–391.

[26] E. C. Neto, D. A. da Costa, and U. Kulesza, "The impact of refactoring changes on the szz algorithm: An empirical study," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2018, pp. 380–390.

[27] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[28] H. Tong, B. Liu, and S. Wang, "Kernel spectral embedding transfer ensemble for heterogeneous defect prediction," *IEEE Transactions on Software Engineering*, 2019.

[29] S. Watanabe, H. Kaiya, and K. Kaijiri, "Adapting a fault prediction model to allow inter languagereuse," in *Proceedings of the 4th international workshop on Predictor models in software engineering*, 2008, pp. 19–24.

[30] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2016, pp. 87–98.

Acknowledgement

Acknowledgement

Your Acknowledgement should go here.

Author
Jiho Shin