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 17, 2020

Approved by

Jaechang Nam

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 17, 2020

Thesis advisor Jaechang Nam
Prof. Jaechang Nam

Committee Member Hong Charmgil
Prof. Charmgil Hong

Committee Member 홍신
Prof. Shin Hong

ICE Jiho Shin 신지호
21931006 Actionable Defect Prediction

즉시 조치 가능한 버그 예측

Department of Information and Communication Engineering,
2020, 39

Advisor: Prof. Jaechang Nam

Abstract

Defect prediction studies have been actively conducted over the past decades. However, existing defect prediction models face challenges such as the lack of actionable messages and the cold-start problem. To overcome these issues, various approaches, i.e. cross-project defect prediction, unsupervised defect prediction, and just-in-time defect prediction, have been studied. However, these approaches are still limited in several aspects. We propose a novel approach to predict defects in the change level by searching for similar changes within the existing software repositories. The model identifies a change as "buggy" if the change is similar to the existing bug-inducing change; otherwise, it predicts a change as "clean". The model then suggests a corresponding patch as an actionable message. Our approach can provide a new direction to the field while addressing two main issues faced by traditional defect-prediction models.

결함 예측 연구는 지난 수십 년 동안 활발하게 수행되어 왔다. 하지만 기존의 결함 예측 모델은 실행 가능한 메시지 부족 및 콜드 스타트 문제들과 같은 어려움에 직면해 있다. 이러한 문제점을 극복하기 위해 교차 프로젝트 결함 예측, 비지도 결함 예측, 즉각 결함 예측 등 다양한 방법들이 연구되어 왔다. 하지만 이러한 연구들 또한 여러 측면의 한계점을 갖고 있다. 따라서, 우리는 이를 해결하기 위해 기존의 없던 유사 수정 기반의 새로운 버그 예측 모델을 제안한다. 이 모델은 예측 대상인 수정이 기존 소프트웨어 저장소에 있는 버그 유발 수정과 유사하다면 버그라 예측하고 유사하지 않으면 클린으로 예측한다. 버그라고 예측된 수정에는 연관된 유사한 패치를 추천하여 실행 가능한 메시지를 제공한다. 본 논문에서 제안하는 모델은 콜드 스타트 및 클래스 불균형 문제를 완화하고 실행 가능한 메시지에 대한 패치를 제안함으로써 결함 예측 모델의 세 가지 문제를 해결할 수 있는 가능성을 보여준다.

Abstract v

List of Figures viii

List of Tables ix

I. Introduction 1
1.1. Main Issues of Defect Prediction 1
1.1.1 Actionable Messages 1
1.1.2 Cold-start Problem 2
1.1.3 Contribution 2

II. Related Work 4
2.1. Similar Commit Search 4
2.1.1 Code Clone Detection/Search 4
2.1.2 Code Search Engine 5
2.1.3 Commit Clustering 5
2.2. Defect Prediction 6
2.2.1 Traditional Defect 6
2.2.2 Cross-Project Defect Prediction 7
2.2.3 Heterogeneous Defect Prediction 7
2.2.4 Just-in-Time Defect Prediction 7
2.3. Automatic Program Repair 8

III. Preliminary Studies 10
3.1. Count-based Change Vector 10
3.2. String Comparison 11

IV. Approach 14
4.1. Data Collection 14
4.2. Vector Embedding 15
4.3. `SimFin`: Similar Change Finder 16
4.3.1 Number of `SimFins` 16
4.3.2 Auto Encoder-Decoder Model 17
4.3.3 k-Nearest Neighborhood 17
4.4. Prediction 18

V. Experimental Setup 20

5.1. Research Questions 20

5.2. Dataset 20

5.3. Baseline 21

5.3.1 Evaluation Metrics 23

VI. Experiment Results 25

6.1. RQ1: `SimFinMo` vs Baseline 25

6.2. RQ2: How actionable is `SimFinMo`? 26

6.3. RQ3: `SimFinMo` with different cut-off values 29

6.4. RQ4: Combined `SimFin` vs. Divided `SimFin` 29

VII.Discussions 31

7.1. Importance of this Approach 31

7.2. Future Work 32

VIIIThreats to Validity 34

8.1. Construct Validity 34

8.2. External Validity 34

8.3. Internal Validity 34

IX. Conclusion 35

X. References 36

Figure 1. A change instance in project isis. 11

Figure 2. A change instance in project ignite. 11

Figure 3. Count-based change vector if Fig. 1 and 2 11

Figure 4. A change instance in project lucene-solr (deleted line) 12

Figure 5. A change in project lucene-solr (added line) 13

Figure 6. A change instance in project oozie (deleted line) 13

Figure 7. A change instance in project oozie (added line) 13

Figure 8. Overall structure of the `SimFin` and `SimFinMo` approach. 14

Figure 9. An example of a direct hint in labeling suggested patches. 28

Figure 10. An example of a indirect hint in labeling suggested patches. 28

Figure 11. An example of a ground truth BFC from project tez. 33

Figure 12. An example of suggested BFC from the project hbase. 33

Table 1. The list of project used as a test set 21

Table 2. Confusion matrix 24

Table 3. The precision value of each baseline and `SimFinMo` 25

Table 4. Recall value of each baseline and `SimFinMo`. 26

Table 5. F1 score of each basline and `SimFinMo` 26

Table 6. MCC of each baseline and `SimFinMo` 26

Table 7. A table that shows the statistics of suggested patches and their labeling. 29

Table 8. This table shows different performance metrics using different cut-off values. The following result is from the project maven. 30

Table 9. This table shows prediction performance of combined `SimFin` and two divided `SimFin` 30

I. Introduction

Along with the rise in software complexity, the cost of quality assurance and software development continues to rise too. 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 automatic program repair (APR) [15, 25, 28], automated test case generation [1, 2, 21] defect detection [36, 37, 43], software defect prediction (SDP) [33, 44, 48]. Among them, (SDP) models have been actively studied to efficiently allocate testing resources to reduce development cost. Software defect prediction uses static code metrics as features on machine learning predictors to identify a module of code as buggy or clean. However, like other techniques, defect prediction models face numerous challenges in practical usage.

1.1. Main Issues of Defect Prediction

There are several issues in SDP such as identifying the relationship between feature metrics and the label, no consistency in adopting performance evaluation metrics, difficulty in gathering buggy data, lack of standard or general framework, lack of validity in the economic benefit, and the validity of how to retrieve buggy data [3, 10]. Apart from these issues, we want to tackle some other important issues that challenge the research field of defect prediction.

1.1.1. Actionable Messages

One of the major limitations that SDP models face is that the predicted results lack actionable or explainable messages for the developers to act upon [22]. The traditional defect prediction model predicts source code modules, 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 module 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 defect prediction (JIT-DP) got into attention [13]. JIT-DP 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 solve the problem, because if the commit gets too big, the same problem occurs. Besides, the approach still does not explain how the commit is causing problems to the software. In other words, the models do not provide actionable messages for the bug-prone changes.

1.1.2. Cold-start Problem

The second major problem of traditional defect prediction models is the cold-start problem. A cold-start problem occurs when the target system lacks historical data [38]. Because traditional defect prediction models are built with previous versions of the target system, it is impossible to apply to 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 on newly starting projects because the prediction models are trained from other existing projects (cross-project). HDP enables CPDP even when the source project and the test project have different metrics as features. Machine learning techniques can only be applied when the training data and the test data have the same dimension of features (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 [13].

1.1.3. Contribution

Herein, we propose a novel and potential SDP paradigm to resolve these issues. Essentially, we search for changes similar to the target change from software repositories and then use their distance value to identify the target change as buggy or clean. We name this model the similar commit change finder (SimFin), which searches for similar changes. We also develop a SimFin-based defect-prediction model (SimFinMo) which alleviates the lack of actionable messages by providing patch suggestions. The intuition of SimFin is as follows. If the target change 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. Also, we do not need a certain period of historical data to collect metrics for test data. For prediction, SimFinMo simply looks 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 the two aforementioned limitations of SDP models.

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 suggesting patches that are used to fix the searched similar change.
3. We mitigate the cold-start problem because our model is fully universal, and needs no single previous commit for a test instance because we do not use historical metrics.

II. Related Work

2.1. Similar Commit Search

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

2.1.1. Code Clone Detection/Search

Jiang et al. [11] proposed an approach named DECARD which represents code blocks as subtrees and uses similarity algorithms on tree data structures. Lee et al. [20] proposed a method that uses a multi-dimensional indexing technique and kNN (k-Nearest Neighborhood) algorithm to reduce the search time while maintaining the functionality of finding semantically similar code fragments. They used 54 MLOC of code to make this code clone detection module. Keivanloo et al. [14] did a similar research, but their main difference is that they used hash tables and binary search algorithms in implementing a multi-level indexing technique. They experimented and evaluated on 266 MLOC of codebases. White et al. [47] 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 codebases. 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.

2.1.2. Code Search Engine

Bajracharya et al. [4] proposed a tool named Sourcerer which searches for code fragments. The tool divides target code fragments concerning the code used to improve the search rate. They have divided the categories into implementation, uses, and structures. McMillan et al. [27] proposed Exemplar (Executable examples archive) helps find code fragments that function 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. [16] proposed a tool that when a user searches for an API document, returns code snippets that can be helpful for the API's usage together with the document. Kim et al. [17] proposed FACOY (Find A Code Other than Yours) searches for a code fragment that is similar to the user's input but syntactically or semantically but with a similar function. Gu et al. [9] 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 by mapping both natural language and code snippet in high dimensional vector space and trains a deep learning model to map these instances as close to space if they have semantically similar.

The difference between code search and commit search is that static code and commit has a different structure, just as code clone search. Second, code search techniques are more focused on handling natural language queries 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 codebases as input to search for commits because commits can be very long.

2.1.3. Commit Clustering

Kreutzer et al. [19] did a study about clustering similar commits concerning 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 the 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. [7] did similar work but with different scope. They categorized different changes to commit concerning 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, naïve Bayes, etc.) for classification, and applied hierarchical clustering to cluster them.

The difference between commit clustering and commit search is that in [19], the clusterings are too big to find the syntactical or semantical similarities of each commit. And as for [7], 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 SDP methods and explain how they are different from each other and our work.

2.2.1. Traditional Defect

Traditional defect prediction predicts a module in different granularity as buggy or clean. In a traditional defect prediction scenario, the granularity is usually in the file-level or the method level. They use the previous version of their project to predict the current or later version of the project. Munson et al. [30] built a classification model to classify if a module has a high risk or not with an accuracy of 92%. Chidamber and Kemerer [6] proposed a suite of object-oriented related metrics that could be applied in defect prediction. Nagappan and Ball [32] proposed code churn metrics to predict the defect density of the system. This was the first process-related metrics and more process-related metrics were proposed after.

2.2.2. 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 a 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. [45] proposed the first CPDP approach to apply prediction models that are already built for other projects. Ma et al. [26] proposed Transfer Naive Bayes (TNB) that weights source instances similar to the target instances Nam et al. [34] proposed TCA+ to alleviate the feature differencing problem in applying CPDP.

2.2.3. Heterogeneous Defect Prediction

Heterogeneous defect prediction was first proposed by [33]. It is a cross-project defect prediction where the source project and the target project have different feature spaces. This method enables the source project to have a different set of features which was an impossible thing to do. With this technique, it expanded the range of projects to be selected as a training set, which is very important because collecting buggy data is very hard. Li et al. [23] proposed cost-sensitive transfer kernel canonical correlation analysis (CTKCCA) to evaluate the nonlinear correlation relationship of the different features. Li et al. [24] proposed a two-staged ensemble learning (TSEL) approach for HDP, which contains an ensemble multi-kernel domain adaptation stage and ensemble data sampling stage. These stages handle separate nonlinear correlation of the features and the imbalance class of the labels. Tong et al. [41] 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.

2.2.4. 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 codebase. Mockus et al. [29] proposed the first to identify changes concerning their specific reasons of causes: adding new features, correcting faults, and restructuring code for future changes. Kim et al. [18] is the first study that did a machine learning modeling for a buggy change of a project. Kamei et al. [12] empirically evaluated the JIT-DP model in the context of a cross-project scenario. They found that the models improve performance when selecting models that use other similar projects, using a larger pool of datasets, 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 prediction. Even though we use autoencoder and kNN, it is for searching for similar commits. The predictions are made by comparing the distance and cutting them with a threshold value.

2.3. Automatic Program Repair

Automatic program repair is a method that automatically generates a bug-fixing change, i.e. a patch, for a defective code. We have added APR in the related studies because it could be related to our patch suggestion method.

Weimer et al. [46] experimented on generating patches using genetic programming (GP) and repeated testing and generating until the test case passed. Kim et al. [15] also used GP in generating patches, however, they pointed out that existing methods purely generated patches randomly. So they did not generate their patches in pure randomness, but they generated them by retrieving patches that are made by developers.

The reason why patch suggestion is important even though there is a research field that generates patches is that APR has limited patch templates to generate. Also, the generated patches must be applied to every test case which takes a very long time. And because the generated patches are validated only on test cases, the patches are overfitted on the test cases. The patch might pass every test case but it is not guaranteed to be a correct patch. Our patch suggestion method has the potential to improve APR techniques by providing mutation operators and code ingredients.

III. Preliminary Studies

This chapter explains the preliminary studies that were conducted to find different change representations. Deciding how to represent changes is very important because the information that can be carried in a representation is very different concerning their data forms.

3.1. Count-based Change Vector

Count-based change vector representation of change has a fixed size of the sparse vector. Each index represents a change type of an AST node. If a certain AST node is added, the index that corresponds to the kind of change is incremented. We capture the changed AST nodes in 4 change types, i.e. addition, deletion, updates, moves, by using a fine-grained source code differencing algorithm, Gumtree [8]. Because the change is represented in a numeric vector, it is suitable to be applied on numerous distance-calculating equations. Specifically, we used the Jaccard index to find similar changes [5].

Fig. 3 shows an example of two count-based change vectors that have the Jaccard index of 0.92. Fig. 1 and 2 show an example of the corresponding changes. As depicted in the figures, we can see that both changes are similar in terms of their AST structures. The underlined values 146 1 and 159 1 in Fig. 3 means that both changes have one importDeclaration (146) and MethodInvocation (159) node added.

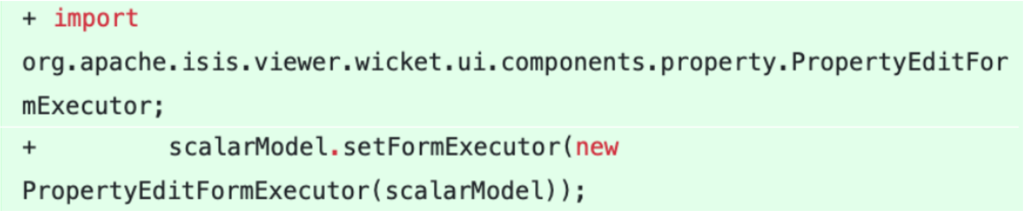
However, there are limitations to this representation. Because the changes are composed of changed AST node counts, it is not able to capture the order of the changed nodes. So changes that had very similar vectors also resulted in totally different looking changes because of different occurrences of nodes. So to capture a more accurate representation of change, we needed a different design.

3.2. String Comparison

To capture the order information of changes, we experimented on using the change code as a string value and used string comparing algorithms in finding similar changes.

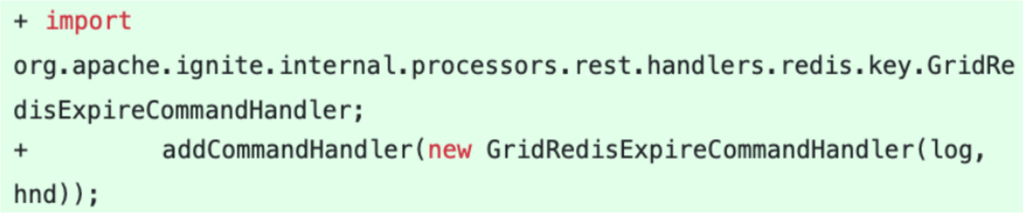
First, we used the traditional line differencing algorithm used in git. We didn't use Gumtree this time because Gumtree only considered the changed node without its context. By retrieving the whole line, we could represent the change with order and context.

To reduce noise information, we only retrieved changes that both deleted lines (buggy) and added lines (patch) within the same method scope. This way we can be sure that the change is a bug-fixing change. Other changes that only have either added or deleted lines are not considered.



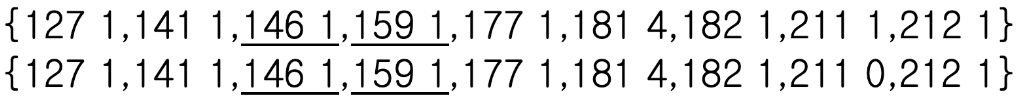
```
+ import org.apache.isis.viewer.wicket.ui.components.property.PropertyEditFormExecutor;
+         scalarModel.setFormExecutor(new PropertyEditFormExecutor(scalarModel));
```

Figure 1: A change instance in project isis.



```
+ import org.apache.ignite.internal.processors.rest.handlers.redis.key.GridRedisExpireCommandHandler;
+         addCommandHandler(new GridRedisExpireCommandHandler(log, hnd));
```

Figure 2: A change instance in project ignite.



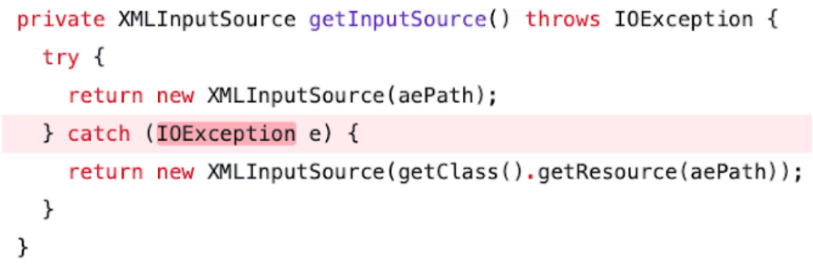
{127 1,141 1,146 1,159 1,177 1,181 4,182 1,211 1,212 1}
{127 1,141 1,146 1,159 1,177 1,181 4,182 1,211 0,212 1}

Figure 3: Count-based change vector if Fig. 1 and 2

Comparing string values take much more time than applying them to numeric values. So we decided to apply the SimHash algorithm to reduce the dimension and skipped comparing if they were over a certain Hamming distance [42]. This method significantly reduced the time of similarity comparison. We also used a mix of Jaccard index and Levenshtein distance to capture both ordering and the collective occurrence of string values.

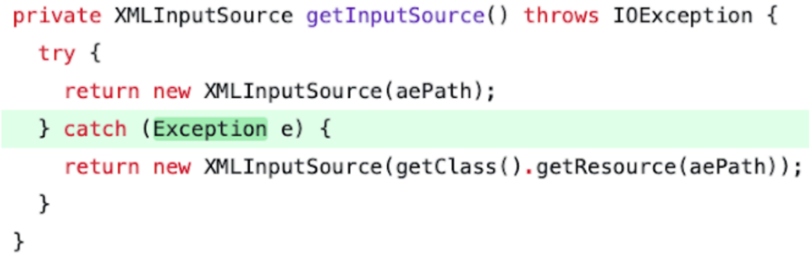
Fig. 4-7 shows an example that had a Jaccard index of 0.86. As depicted in the figures, changes that had the highest similarity values were usually very short. Some changes were longer, but they had a low similarity value. This result seemed natural because when string values are longer there are more chances that they will be different. So we decided on using a different method which is the `SimFin` to find similar commits in the software repositories.

For the count-based change vector and string comparison, we should find hand-designed features such as the number of AST node occurrences or string similarities. Also, the features are limited to syntactic characteristics of code changes. Therefore, we propose a method to find similar commits by applying deep learning so that the computer can directly derive semantic features.



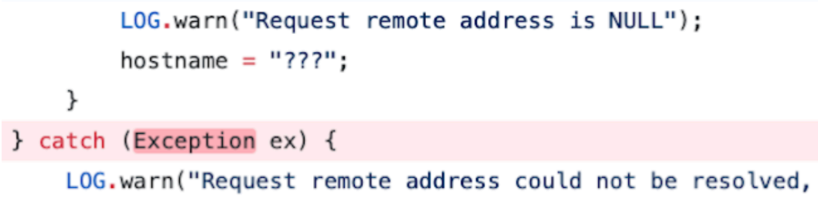
```
private XMLInputSource getInputSource() throws IOException {
  try {
    return new XMLInputSource(aePath);
  } catch (IOException e) {
    return new XMLInputSource(getClass().getResource(aePath));
  }
}
```

Figure 4: A change instance in project lucene-solr (deleted line)



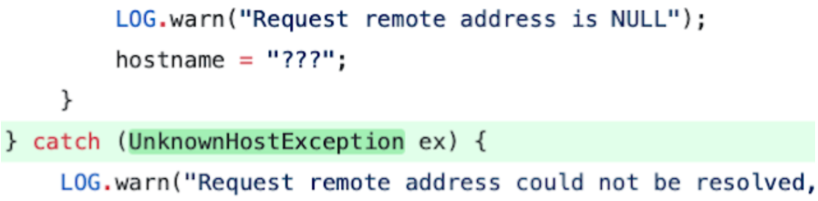
```
private XMLInputSource getInputSource() throws IOException {
  try {
    return new XMLInputSource(aePath);
  } catch (Exception e) {
    return new XMLInputSource(getClass().getResource(aePath));
  }
}
```

Figure 5: A change in project lucene-solr (added line)



```
        LOG.warn("Request remote address is NULL");
        hostname = "???";
    }
} catch (Exception ex) {
    LOG.warn("Request remote address could not be resolved,
```

Figure 6: A change instance in project oozie (deleted line)



```
        LOG.warn("Request remote address is NULL");
        hostname = "???";
    }
} catch (UnknownHostException ex) {
    LOG.warn("Request remote address could not be resolved,
```

Figure 7: A change instance in project oozie (added line)

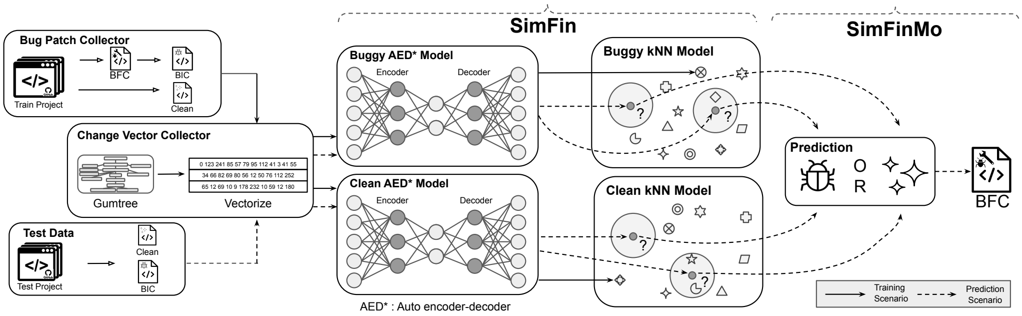


Figure 8: Overall structure of the `SimFin` and `SimFinMo` approach.

IV. Approach

This chapter details the implementation of `SimFinMo`. The overall approach of `SimFinMo` is illustrated in Fig. 8.

4.1. Data Collection

We mined bug-inducing commits (BICs) and clean commits from software repositories to train the `SimFin`. First, we used the SZZ algorithm [39] to collect BICs from software repositories. The SZZ algorithm first mined bug fixing changes (BFC) from an 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 labeled as the following: types that are labeled “Bug”, statuses that are labeled “Closed” or “Resolved” and resolutions that are labeled “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. The deleted lines in the BFC are considered to be the buggy code while the added lines are the corresponding patches. We compare the 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 the Myers diff algorithm [31]. Then, we apply git the blame command to each modified line since git blame shows the information of the last commit id, the author, the timestamp, and the line number of code for the line. By using this piece of information, we mined the BIC.

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

4.2. Vector Embedding

We vectorized the change data after mining them from software repositories First, we collect the source code of a change applied to the Gumtree [8] algorithm. 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 fewer false-positive changes (code parts that are not 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 the 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. By doing this, we can capture the syntactic characteristic as well as the order information of the change. After that, we append what we call the context vector. The context vector is a list of neighboring nodes of each node-change. We collect the context vectors by taking all the descendant nodes of the parent node of each change 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 several redundant context nodes. We have chosen to use the context vector to capture 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 has 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 a way is because when `SimFin` is given a target commit, it identifies the closest commit in software repositories. The key value should contain the id of the commit and the source file path of the change to return the most similar commit.

4.3. `SimFin`: Similar Change Finder

`SimFin` is composed of a deep auto encoder-decoder (AED) and a k-Nearest Neighborhood (kNN) model. The deep AED model first encodes the embedded syntactic vectors from Gumtree and learns the semantics of changes. Then the kNN model computes distances of each data point of changes and returns the closest *k* changes and its distance.

4.3.1. Number of `SimFins`

First, we have built one `SimFin` where all the BICs and clean changes are trained together. This is reasonable as we are going to use the ratio of the distance value of the closest BIC and the clean change to the target change in identifying the defect. By plotting all the changes in the same encoding space, the degree of distance between the target change, and both clean and buggy changes will have the same degree.

Second, we also built two separate `SimFin` where one pair of AED and kNN model is trained with buggy change instances and the other with clean change instances. When we build two separate `SimFin` models, the encoding space of buggy and clean changes become different, hence using their distance ration can be insensible. However, due to the data size of clean changes and buggy changes being extremely imbalanced (around 26 :

1), looking for the closest buggy and clean changes within a certain *k* can show a very skewed result.

To alleviate these issues raised from the imbalanced class, we trained a separate AED model to get the distance of the closest *k* change instances in each buggy and clean data pool. With *k* numbers of closest distances in both pools, we hope to get the representative distance value of each pool.

For the validity of choosing whether to train one or two different `SimFins`, we compared the performance measure of the two methods in section VI.

4.3.2. Auto Encoder-Decoder Model

AED model in `SimFin` is used to learn and encode the relationship between the syntactic feature and its semantics. First, we apply zero-padding to all the training instances to match the 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 AED model are 5 layers for each network, 500 nodes for each layer, 3 epochs, and a batch size of 256. We used ReLU at each layer 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. The decoder of the network is used to update the weight of the encoder model, but we only use the encoder part of the network to encode the test projects in the prediction phase. The BIC instances and the clean instances are trained into separate AED models. We trained the networks separately because we made a presumption that BIC and clean changes have different characteristics.

4.3.3. 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 predicts an instance's class concerning the distance in the vector space. The labels are usually a binary or multi-class of labels, however, we use the commit key as the label which is a unique label. This is done because we want to search for the closest commit to the target commit. All the label in kNN model, Fig. 8, is depicted as different icons to show that each label is unique. Because of this nature, it is not able to, or not sensible, to get the evaluation score of the kNN. Similar to the AED model, the kNN models are also trained separately from BIC instances and clean instances. Thus, we finally have buggy `SimFin` and clean `SimFin` respectively.

4.4. Prediction

In the prediction phase, the target commit is vectorized and fed into the `SimFin`. The change vectors, together with the context vectors, are generated from applying the Gumtree algorithm. The semantic representation of the change is inferred from the encoder that is trained before-hand. Lastly, the learned feature representation is plotted in the vector space of the kNN model, which then searches for the nearest changes. Because we have a 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 and is 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 the closest distance value from buggy `SimFin` while δ_c is 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 `SimFinMo` suggests the 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.

V. Experimental Setup

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

5.1. Research Questions

We considered two research questions to evaluate the defect prediction of `SimFinMo`.

- RQ1: Was the defect prediction of `SimFinMo` potentially comparable to those of various machine learners?
- RQ2: How actionable are patches suggested by the `SimFinMo`?
- RQ3: What were the impacts of various `SimFinMo` cutoffs on prediction performance?
- RQ4: How effective is using the divided `SimFin` in terms of predictive performance?

We investigated the effectiveness of `SimFinMo` by comparing its prediction performance with the existing baseline. Furthermore, we studied various aspects of `SimFinMo` using different cut-off values.

5.2. Dataset

We used 193 active, Java projects in the Apache Software Foundation (ASF) to construct `SimFin`. We chose the projects that maintain active GitHub- or Jira-issued tracking systems. A total of 110K BIC instances and 2.6M clean instances were collected for the training data to build the buggy and clean `SimFin` models.

Table 1 shows the details of the test project. The test set used are also from ASF. These test projects were selected by considering various buggy ratios and the different number of commits. JIT-DP conducted at the change level remains very challenging to achieve high prediction performance. One of the reasons is that the number of buggy commits is significantly smaller than that of clean commits. These ratios are affected by the total number of commits in a project. Thus, for our test data, we randomly chose six 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.

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

Name	# of Buggy	# of Clean	Total
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

5.3. Baseline

The baseline we use to compare the prediction performance is a typical JIT-DP metric reported in Kamei et al. [13]. The metric types are classified into five dimensions: diffusion, size, purpose, history,and experience.

The **diffusion dimension** of a change shows how distributed a change is. A distributed change can be measured by counting the different components of source files. There are four features in this category: number of modified subsystems (NS), number of modified directories (ND), number of modified files (NF), and distribution of modified code(Entropy). For our experiment, we considered the number of the subsystem as one because they are all Java projects. For illustration, if a commit changed three files: java/src/clami/main.java, java/src/clami/utils.java, and the java/src/remi/input.java. NS is one (java/src/), ND is two (clami/ and city/) and NF is three (main.java, input.java, and utils.java). Entropy counts the distribution of modified lines in a source file. **Size**

dimension is the number of changed lines in a source file. The three features of these categories are lines of code added (LA), lines of code deleted (LD), and lines of code in a file before the change (LT). The **purpose dimension** has one feature that which is FIX. FIX is a binary feature that labels if a commit is a BFC or not. This is used as a feature because a BFC is more likely to introduce new bugs. The **history dimension** is about the revision history of changes from the past to the present. NDEV is the number of unique developers who have modified a source file. AGE is the time between the current source and the most recent modification. 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 three (α, β, and γ). The **experience dimension** is the information of developers in the project. Developers who frequently participate in the project is less likely to cause bugs because the developer understands the project well. The experience dimension has two factors: 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 commits in 2018, and two commits 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., $\frac{2}{2} + \frac{3}{4} + \frac{1}{5}$).

With these metrics, an online change classification model is built as reported from Tan et al. [40]. Online change classification is proposed to address the limitation of cross-validation on change classification. Firstly, cross-validation is not suitable in change classification because it will use future data for predicting bugs from the past. This scenario does not match the real-world scenario because buggy change and a fixing change has a chronological relationship. Secondly, there is a chance of mislabeling the changes. When we predict a buggy change in a certain period, the buggy change is labeled as buggy because there is a bug-fixing change in the future. However, in the time the fixing didn't happen yet so it should be labeled as clean.

To address these issues, Tan et al. [40] proposed an online change classification method. They left a gap between the training data and the test data. This gap should be an estimate of bug-fixing time so that there is enough time for the buggy changes in the trainset to be discovered. Also, the method is to construct an online model, meaning that the training data is accumulated and updated as time goes by. By doing this, the model is applied with multiple runs alleviating the sensitivity that can be dependent on a particular time.

In online change classification, the first two datasets cannot be used as a test set because when predicting the first dataset, the prediction time is before any bug-fixing is done and therefore there will be no buggy changes to predict from. The second data set cannot be used as well because at least one period has to skip to leave a gap. For a fair comparison, the evaluation for `SimFinMo` does not predict the first two sets of data.

We have assessed six of the most used machine learning algorithms in defect prediction. The algorithms we trained are BayesNet (BN), k-Nearest Neighbor (IBk), Logistic model tree (LMT), Naive Bayes (NB), and random forest (RF).

5.3.1. Evaluation Metrics

The evaluation metrics for comparing `SimFinMo` and baseline are precision, recall, F1 score and MCC. We used a variety of evaluation metrics to assess a sound experiment and show various aspects of the predictors. A Confusion matrix is needed to evaluate two models. There are four metrics as shown in table 2. True positive (TP) is when the actual label is true and the model predicts it as true. False positive (FP) is when the actual label is false but the model predicts as true. False negative (FN) is when the actual label is true but the model predicts as false. True negative (TN) is when the actual label is false and the model predicts it as false. Precision is the 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 ($Recall = \frac{TP}{TP+FN}$). F1-score is the harmonic mean of the precision and recall ($F1 - score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$). MCC is the Matthews correlation coefficient that is a measure used in unbalanced labels ($MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$.

Table 2: Confusion matrix

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

VI. Experiment Results

This chapter shows the experiment results of the reproduced baseline and our approach of SimFinMo. Table 3- 6 shows the overall results of the baseline and our approach. The bold values in the table indicate the highest score from the projects.

6.1. RQ1: SimFinMo vs Baseline

From Table 3, we can see that all most all of the baseline results are better in precision. However, Table 4 shows that SimFinMo always has better performance in the 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 Table 5, we can see that out of 6 projects, SimFinMo outperforms 4 of the projects in F1 score. The average F1 score of SimFinMo is also the highest out of all the baseline machine learning algorithms. Table 6 shows a mixed result, but overall, random forest shows the best performance in terms of MCC with the highest average. From the results, we can state that SimFinMo outperforms the baseline overall.

Table 3: The precision value of each baseline and SimFinMo

Name	BN	IBk	J48	LMT	NB	RF	SFM
maven	0.220	0.141	0.253	0.297	0.147	**0.362**	0.146
ranger	0.200	0.144	0.162	0.141	0.182	**0.350**	0.183
sentry	0.154	0.065	0.168	0.160	0.098	**0.233**	0.141
sqoop	0.220	0.153	0.205	0.243	0.170	**0.379**	0.036
syncope	0.268	0.099	0.209	0.200	0.076	**0.280**	0.056
tez	0.161	0.173	0.232	0.159	0.150	**0.265**	0.241
average	0.204	0.129	0.205	0.200	0.137	**0.311**	0.134

Table 4: Recall value of each baseline and SimFinMo.

Name	BN	IBk	J48	LMT	NB	RF	SFM
maven	0.114	0.190	0.126	0.103	0.322	0.065	**0.834**
ranger	0.539	0.162	0.267	0.278	0.831	0.125	**0.886**
sentry	0.294	0.120	0.160	0.184	0.341	0.104	**0.887**
sqoop	0.207	0.193	0.213	0.261	0.441	0.107	**0.846**
syncope	0.023	0.078	0.040	0.009	0.172	0.019	**0.877**
tez	0.298	0.355	0.371	0.528	0.665	0.302	**0.841**
average	0.246	0.183	0.196	0.227	0.462	0.120	**0.862**

Table 5: F1 score of each basline and SimFinMo

Name	BN	IBk	J48	LMT	NB	RF	SFM
maven	0.150	0.162	0.168	0.153	0.202	0.111	**0.248**
ranger	0.292	0.152	0.202	0.187	0.299	0.184	**0.303**
sentry	0.202	0.085	0.164	0.171	0.153	0.143	**0.243**
sqoop	0.213	0.171	0.209	**0.251**	0.245	0.167	0.068
syncope	0.042	0.087	0.068	0.017	**0.106**	0.036	0.105
tez	0.209	0.232	0.285	0.244	0.244	0.282	**0.375**
average	0.185	0.148	0.183	0.170	0.208	0.154	**0.224**

Table 6: MCC of each baseline and SimFinMo

Name	BN	IBk	J48	LMT	NB	RF	SFM
maven	0.096	0.058	0.119	**0.126**	0.089	0.120	0.056
ranger	0.150	0.015	0.047	0.017	**0.182**	0.143	0.095
sentry	0.098	-0.043	0.080	0.079	0.013	0.100	**0.101**
sqoop	0.084	0.011	0.071	0.119	0.053	**0.137**	0.047
syncope	0.059	0.021	**0.061**	0.027	-0.001	0.056	0.051
tez	0.051	0.076	0.155	0.074	0.068	**0.167**	0.033
average	0.089	0.023	0.089	0.074	0.067	**0.121**	0.064

6.2. RQ2: How actionable is SimFinMo?

To show how much the patches that are suggested by SimFinMo are helpful to developers, we manually inspected the patches and labeled each of them. We only inspected patches that were predicted to be a true positive. We also only considered patches where the ground truth patches are short. The reason we only considered short patches because most of the longer patches are composed of refactoring and architecture changes. With these kinds of patches, it is difficult to analyze the patches correctly, due to the variety of number and domain of the used projects. The labels are as follows:

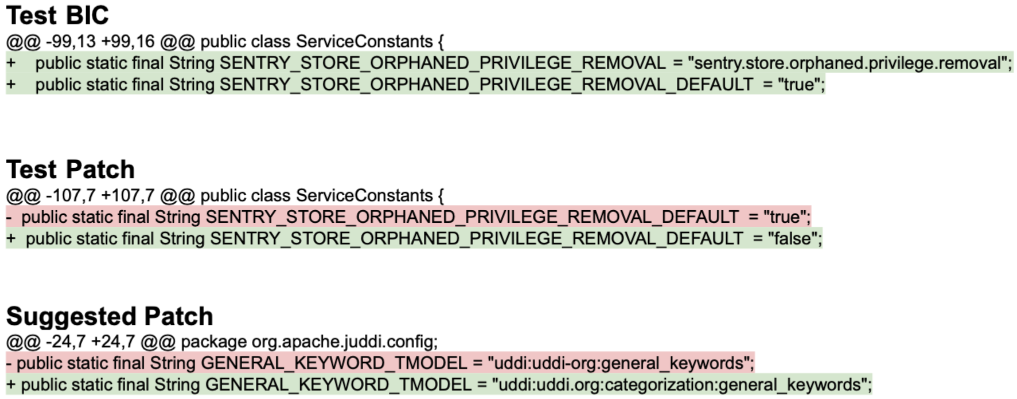
1. **Direct hint**: the ground truth patch and the suggested patch are the same changes to their AST node type.
2. **Indirect hint**: the ground truth patch and the suggested patch are not the same changes to their AST node type, however, they have similar node types with high relevance.
3. **No hint**: the ground truth patch and the suggested patch does not have the same changes to their AST node type and has no relevance at all.

The labeling can be very subjective to say that changes have relevance or not. So we include some examples to show how the manual labels are done.

In Fig. 9, there are three changes: the target BIC change, the ground truth patch of the target BIC change, and the suggested patch. By looking at the ground truth patch, we can see that the initialization of a string variable contains a bug as it is replaced with another value (true -> false). While they have different values, we can see that the suggested patch also changes a public static final String value. In cases where the changed AST node types are the same between the ground truth patch and the suggested patch, we label them as *Direct hint*s.

A Similar example is shown in Fig. 10. The introduced bug has to do with omitting a parameter in a method implementation. The suggested patch also shows that it added a method parameter. However, it is happening in a method invocation line of code. And the type of the changed parameter is most likely to be a different type. Even though the change contains different types of AST nodes, we can still infer that the omission of a parameter can help fix the bug. In these kinds of cases, we label them as *Indirect hint*s.

Through this method of labeling, we inspected all the test projects. Table 7 shows the



Test BIC

```
@@ -99,13 +99,16 @@ public class ServiceConstants {
+    public static final String SENTRY_STORE_ORPHANED_PRIVILEGE_REMOVAL = "sentry.store.orphaned.privilege.removal";
+    public static final String SENTRY_STORE_ORPHANED_PRIVILEGE_REMOVAL_DEFAULT  = "true";
```

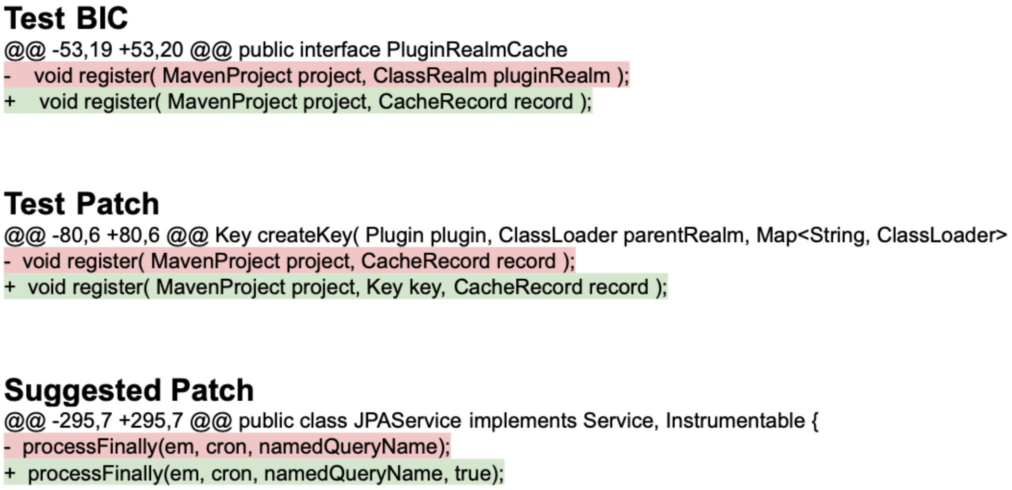
Test Patch

```
@@ -107,7 +107,7 @@ public class ServiceConstants {
-  public static final String SENTRY_STORE_ORPHANED_PRIVILEGE_REMOVAL_DEFAULT  = "true";
+  public static final String SENTRY_STORE_ORPHANED_PRIVILEGE_REMOVAL_DEFAULT  = "false";
```

Suggested Patch

```
@@ -24,7 +24,7 @@ package org.apache.juddi.config;
- public static final String GENERAL_KEYWORD_TMODEL = "uddi:uddi-org:general_keywords";
+ public static final String GENERAL_KEYWORD_TMODEL = "uddi:uddi.org:categorization:general_keywords";
```

Figure 9: An example of a direct hint in labeling suggested patches.



Test BIC

```
@@ -53,19 +53,20 @@ public interface PluginRealmCache
-   void register( MavenProject project, ClassRealm pluginRealm );
+   void register( MavenProject project, CacheRecord record );
```

Test Patch

```
@@ -80,6 +80,6 @@ Key createKey( Plugin plugin, ClassLoader parentRealm, Map<String, ClassLoader>
-  void register( MavenProject project, CacheRecord record );
+  void register( MavenProject project, Key key, CacheRecord record );
```

Suggested Patch

```
@@ -295,7 +295,7 @@ public class JPAService implements Service, Instrumentable {
-  processFinally(em, cron, namedQueryName);
+  processFinally(em, cron, namedQueryName, true);
```

Figure 10: An example of a indirect hint in labeling suggested patches.

statistics of the suggested patch labeling. We can see that 28% ˜60% of the patches are a direct or indirect hint. The percentage of the hint concerning the suggested patch can be shown as low. However, it is encouraging to see this result because no other method provided an actionable message this direct in other defect prediction methods. And to consider that the result only shows the top 1 suggestions, we can hope that the hint rate will be higher when we consider more top k instances.

Table 7: A table that shows the statistics of suggested patches and their labeling.

Name	Direct hint	Indirect hint	No hint
maven	6 (18%)	8 (23%)	20 (59%)
ranger	11 (16%)	16 (24%)	40 (60%)
sentry	4 (14%)	7 (24%)	18 (62%)
sqoop	3 (30%)	3 (30%)	4 (40%)
syncope	9 (12%)	14 (19%)	52 (69%)
tez	10 (14%)	10 (14%)	70 (72%)

6.3. RQ3: `SimFinMo` with different cut-off values

To provide a better concept of how well the model `SimFinMo` predicts defective modules, we have investigated the different performance values with different cut-off values. The results are tabulated in Table 8. Due to the limitation of space in the report, we have only tabulated one of the test projects, maven. From the table, we can see that the precision score is highest when the cut-off value is low (closer 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. However, recall starts to go up rapidly as the cut-off value gets higher. The ascending of the recall is much higher than the descending of the precision yielding a good value of f1-score.

6.4. RQ4: Combined `SimFin` vs. Divided `SimFin`

In section IV, we have talked about how we decided on choosing one combined design of `SimFin` or a buggy/clean divided design of `SimFin`. To show how different designs of `SimFin` contribute to the prediction performance of `SimFinMo`, we compare their result. In table 9, we can see that the divided `SimFin` outperforms the combined `SimFin` in precision, recall, and f1-score of every test project. It also outperforms 4 out of 6 test projects in MCC. Through this result, we think that dividing the encoding space of buggy and clean changes helps to retrieve the representative distance from the target

Table 8: This table shows different performance metrics using different cut-off values. The following result is from the project maven.

Cut-off	Precision	Recall	F1 Score	MCC
0.1	**0.286**	0.002	0.004	0.014
0.2	0.167	0.003	0.006	0.005
0.3	0.183	0.036	0.061	0.023
0.4	0.190	0.146	0.165	0.054
0.5	0.196	0.351	0.252	0.100
0.6	0.194	0.531	**0.284**	**0.131**
0.7	0.177	0.650	0.278	0.120
0.8	0.155	0.727	0.255	0.073
0.9	0.149	0.797	0.250	0.062
1	0.146	0.837	0.248	0.056
2	0.144	0.924	0.249	0.063
3	0.144	de0.934	0.250	0.066
4	0.144	0.936	0.250	0.066
5	0.144	0.939	0.250	0.068
6	0.144	0.940	0.250	0.067
7	0.143	0.940	0.249	0.064
8	0.143	0.940	0.248	0.062
9	0.143	0.942	0.249	0.064
10	0.143	**0.942**	0.249	0.064

change.

Table 9: This table shows prediction performance of combined `SimFin` and two divided `SimFin`

	combined `SimFin`				divided `SimFin`			
Name	Precision	Recall	F1-score	MCC	Precision	Recall	F1-Score	MCC
maven	0.109	0.660	0.187	**0.093**	**0.146**	**0.834**	**0.248**	0.056
ranger	0.123	0.726	0.210	0.060	**0.183**	**0.886**	**0.303**	**0.095**
sentry	0.115	0.709	0.198	0.071	**0.141**	**0.887**	**0.243**	**0.101**
sqoop	0.029	0.495	0.055	0.042	**0.036**	**0.846**	**0.068**	**0.047**
syncope	0.054	0.684	0.100	0.048	**0.056**	**0.877**	**0.105**	**0.051**
tez	0.161	0.570	0.252	**0.058**	**0.241**	**0.841**	**0.375**	0.033

VII. Discussions

In this chapter, we will discuss how this approach is important in software engineering practice, especially in automated quality assurance techniques and future work.

7.1. Importance of this Approach

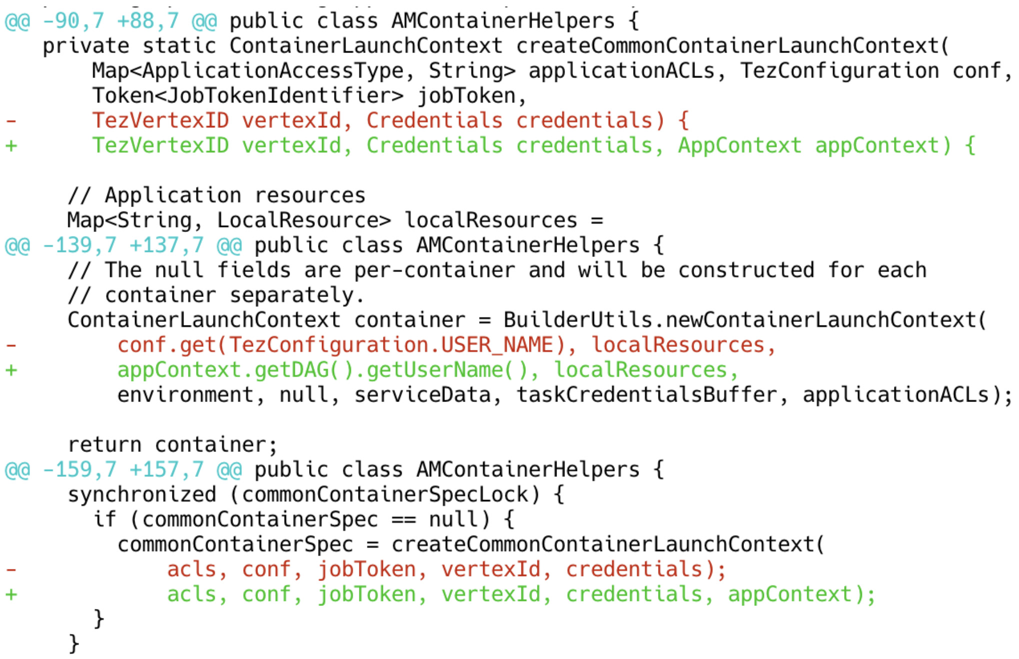
Our defect-prediction approach is novel in that we built a large AED model of 196 projects to encode various change data and use the distance values achieved from the kNN model to make the prediction. Our `SimFinMo` is fully universal which can be fully utilized in a CPDP setting without the need for historical metrics. With this approach, we can say that the cold-start problem is fully alleviated. Another major problem of defect prediction is the lack of actionable messages. Traditional defect-prediction models predict the risk of a module in a file or at a method level. For larger projects, developers need a significant amount of time to find the bug inside the risky module and fix it. JIT-DP was actively studied to resolve the issue because code changes were typically significantly smaller than an entire file or method. By reducing the granularity, developers can easily identify the location of the defect as the code to be inspected is shorter. However, it does not resolve the fundamental problem. JIT-DP does not inform the developers how the defect can be corrected. Using `SimFinMo`, we can identify a risky change and also show the original BFC that is associated with the BIC with the nearest distance value. The suggested BFC can act as an actionable message and provide insights to assist developers in fixing the risky change. Fig. 11 shows an example of a BFC that we collected from project tez, one of the testing projects. Fig. 12 shows an example of a suggested BFC generated by running SimFinMo on the BIC that is shown in Fig. 11. Fig. 12 shows that one parameter for calling a super constructor is added for a change. By looking at Fig. 11, we can see that first and the third change is very similar to the suggested patch in that they are both adding a parameter in a method invocation. Fig. 12 added an AppContext instance as a parameter as its method creates a common container launcher context while

Fig. 3 added an EventType info parameter as it is related to a handler method. Based on the context of a change, developers can get an actionable message such as adding a missing parameter. As the example illustrates, we hope that these suggested patches will help developers to act upon for quality assurance activity. Our method of defect prediction is a novel technique that does not use a 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 shortcomings of machine learning techniques. Cold-start problem is caused because training data is necessary for the model to be built 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 for historical metrics, it fully resolves the cold-start problem.

7.2. Future Work

It is necessary to find the right threshold value to separate clean instances from buggy instances in the future. We used different threshold values for different test projects with which we could only linearly separate the relationship among the classes. However, more studies should be conducted to distinguish the relationship using a more complex and non-linear approach. It was difficult to determine a global threshold that was suitable for all the projects.

Additionally, the suggested BFCs can be used as a code ingredient or mutation operator in automatic program repair (APR). In APR, reducing the search space is critical to find the right patch within a feasible time. Ingredients or a list of operators retrieved from the suggested BFC can help reduce the search space to find the correct patch. So these patch suggestions not only have a high potential for actionable defect prediction but also for more efficient APR technique as well.

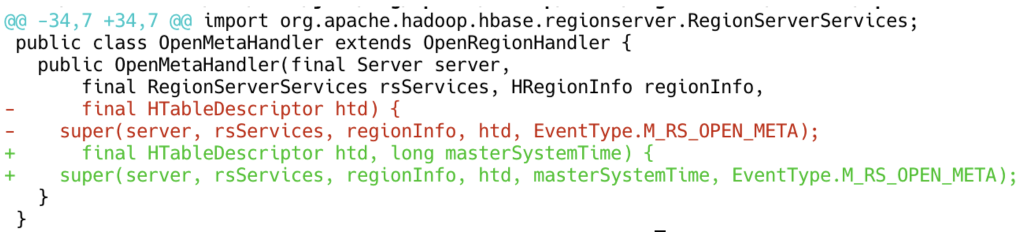


```
@@ -90,7 +88,7 @@ public class AMContainerHelpers {
   private static ContainerLaunchContext createCommonContainerLaunchContext(
       Map<ApplicationAccessType, String> applicationACLs, TezConfiguration conf,
       Token<JobTokenIdentifier> jobToken,
-      TezVertexID vertexId, Credentials credentials) {
+      TezVertexID vertexId, Credentials credentials, AppContext appContext) {

     // Application resources
     Map<String, LocalResource> localResources =
@@ -139,7 +137,7 @@ public class AMContainerHelpers {
     // The null fields are per-container and will be constructed for each
     // container separately.
     ContainerLaunchContext container = BuilderUtils.newContainerLaunchContext(
-        conf.get(TezConfiguration.USER_NAME), localResources,
+        appContext.getDAG().getUserName(), localResources,
         environment, null, serviceData, taskCredentialsBuffer, applicationACLs);

     return container;
@@ -159,7 +157,7 @@ public class AMContainerHelpers {
     synchronized (commonContainerSpecLock) {
       if (commonContainerSpec == null) {
         commonContainerSpec = createCommonContainerLaunchContext(
-            acls, conf, jobToken, vertexId, credentials);
+            acls, conf, jobToken, vertexId, credentials, appContext);
       }
     }
```

Figure 11: An example of a ground truth BFC from project tez.



```
@@ -34,7 +34,7 @@ import org.apache.hadoop.hbase.regionserver.RegionServerServices;
 public class OpenMetaHandler extends OpenRegionHandler {
   public OpenMetaHandler(final Server server,
       final RegionServerServices rsServices, HRegionInfo regionInfo,
-      final HTableDescriptor htd) {
-    super(server, rsServices, regionInfo, htd, EventType.M_RS_OPEN_META);
+      final HTableDescriptor htd, long masterSystemTime) {
+    super(server, rsServices, regionInfo, htd, masterSystemTime, EventType.M_RS_OPEN_META);
   }
 }
```

Figure 12: An example of suggested BFC from the project hbase.

VIII. Threats to Validity

8.1. Construct Validity

In collecting the change vectors, we have disregarded updates and move operations from the changes due to the occurrence of simple refactoring changes with no actual behavioral changes. However, this may lead to missing some important changes such as changing the method invocation to complete another method invocation or moving an if statement to the next line to change the control flow of the program. We believe that this sacrificed some of the true positives to eliminate many of 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`.

8.2. External Validity

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

8.3. Internal Validity

For collecting BIC data we have used the SZZ algorithm [39]. 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 [35]. However, with the circumstances, `SimFinMo` showed the comparable or better performance to the existing baselines. So we believe that the better algorithm that has better precision in BIC instance collecting will improve the performance of `SimFinMo`.

IX. Conclusion

We proposed a novel universal and actionable JIT-DP model that overcomes the main limitations of current defect prediction models, the lack of actionable messages, and the cold-start problem. This study is the first of its kind in that it predicts defective modules at a change level without a machine-learning predictor. Although `SimFinMo` is not perfect, it does surpass some of the most used machine-learning algorithms. Further studies in this direction will contribute to automated quality assurance techniques such as defect prediction and APR. Studies on topics such as finding a better solution to determine a global cut-off value or collecting a higher quality of defects and clean changes will hopefully enhance the prediction performance and contribute to the provision of better actionable messages.

X. References

[1] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, “A systematic review of the application and empirical investigation of search-based test case generation,” *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 742–762, 2009.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino, *et al.*, “An orchestrated survey of methodologies for automated software test case generation,” *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[3] I. Arora, V. Tetarwal, and A. Saha, “Open issues in software defect prediction,” *Procedia Computer Science*, vol. 46, pp. 906–912, 2015.

[4] 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 *OOPSLA*, 2006, pp. 681–682.

[5] S.-H. Cha, “Comprehensive survey on distance/similarity measures between probability density functions,” *City*, vol. 1, no. 2, p. 1, 2007.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] S. Herbold, A. Trautsch, and F. Trautsch, “Issues with szz: An empirical assessment of the state of practice of defect prediction data collection,” *arXiv preprint arXiv:1911.08938*, 2019.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 802–811.

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

[17] 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 *ICSE*, 2018, pp. 946–957.

[18] 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.

[19] 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.

[20] 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.

[21] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "Ipog/ipog-d: Efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008.

[22] 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.

[23] 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.

[24] 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.

[25] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[26] 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.

[27] 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.

[28] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

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

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] M. Pradel and T. R. Gross, "Leveraging test generation and specification mining for automated bug detection without false positives," in *2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 288–298.

[37] M. Pradel and K. Sen, “Deepbugs: A learning approach to name-based bug detection,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[38] A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock, “Methods and metrics for cold-start recommendations,” in *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, 2002, pp. 253–260.

[39] 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.

[40] M. Tan, L. Tan, S. Dara, and C. Mayeux, “Online defect prediction for imbalanced data,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, vol. 2, 2015, pp. 99–108.

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

[42] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, “On the effectiveness of simhash for detecting near-miss clones in large scale software systems,” in *2011 18th Working Conference on Reverse Engineering*, IEEE, 2011, pp. 13–22.

[43] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, “Bugram: Bug detection with n-gram language models,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 708–719.

[44] S. Wang, T. Liu, and L. Tan, “Automatically learning semantic features for defect prediction,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, IEEE, 2016, pp. 297–308.

[45] 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.

[46] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically finding patches using genetic programming,” in *2009 IEEE 31st International Conference on Software Engineering*, IEEE, 2009, pp. 364–374.

[47] 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.

[48] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, “Cross-project defect prediction: A large scale experiment on data vs. domain vs. process,” in *ESEC/FSE*, 2009, pp. 91–100.