ChangeChecker: A Tool for Defect Prediction in Source Code Changes based on Incremental Learning Method

Zi Yuan
Department of Computer Science
Beihang University
Beijing, China
yuanzi@sei.buaa.edu.cn

Lili Yu
Software Testing Center
Second Artillery
Beijing, China
xueer-123@263.net

Chao Liu
Department of Computer Science
Beihang University
Beijing, China

Linghua Zhang
Software Research and Development Center
Oracle
Beijing, China

***Abstract*—In software development process, software developers may introduce defects as they make changes to software projects. Being aware of introduced defects immediately upon the completion of the change would allow software developers or testers to allocate more resources of testing and inspecting on the current risky change timely, which can shorten the process of defect finding and fixing effectively. In this paper, we propose a software tool called ChangeChecker to help software developers predict whether current source code change has any defects or not during the software development process. This tool infers the existence of defect by dynamically mining patterns of the source code changes in the revision history of the software project. It mainly consists of three components: (1) incremental feature collection and transformation, (2) real-time defect prediction for source code changes, and (3) dynamic update of the learning model. The tool has been evaluated in a large famous open source project Eclipse and applied to a real software development scenario.**

Keywords—software engineering; source code change; defect prediction; incremental learning

I. INTRODUCTION

In the past few years, more and more attention has been paid to software quality. Software defects, which affect software quality negatively, are considered to be the key risk and major cost driver for both companies that develop software and companies that consume software systems in their daily business [1]. Normally, if software defects are not fixed timely, they will affect the software in three aspects negatively. Firstly, defects that settle in software for a long time may cause new defects or subsequent error modifications [2]. Secondly, if these defects are discovered and reported late, it will be costly for the developer to reacquaint himself with the changed source code, which will increase the cost of defect fix effort. Thirdly, if these defects are reported by customers after software release, it will damage the software reputation significantly. If there is a tool that can accurately predict whether one or more defects have been introduced immediately after a change is made, it will enable developers to take steps to fix the defects timely.

In this paper, the desired tool called ChangeChecker has been proposed based on machine learning classification method, which takes a source code change as a learning instance, a series of characteristics of it as features, and predicts whether this change introduces defects or not. Given by the data characteristics of the source code changes, there are totally four problems to be handled in building ChangeChecker:

The first problem is which features of a source code change closely related to defects, and how to collect and transform them. The source code changes possess the characteristic of data stream and they constantly arrive with the passing of time. Therefore, the whole data set cannot be obtained by one-time collection and transformation. Incremental way is desirable.

The second problem is that the decision making should be real time. When a new change occurs, the classification result must be given at once. Therefore, it is impractical to rebuild the learning model every time a new change arrives.

The third problem is that due to the variation of underlying defect generation process with time, concept drift is inevitable. Therefore the predictive power of source code change features varies as time: features that in the past have been important, become redundant with the passing of time and new high-predictive features arise that have not been considered before. Therefore, there is a pressing need for a mechanism that can update the learning model dynamically to handle the concept drift.

The fourth problem is that the class distribution of the change data set is imbalanced and unstable. Usually, the number of source code changes that did not introduce defects is much larger than that of the defect introducing changes. But

sometimes the class distribution will vary with the passing of time. Therefore, it is very essential to bring up a mechanism that can mitigate the negative effects of class imbalance on the classification performance.

In order to solve the four problems mentioned above, the ChangeChecker proposed in this paper is mainly made up of three parts: (1) incremental feature collection and discretization, which is used to solve the first problem; (2) real-time defect prediction with incremental Naïve Bayes Classifier, which is used to solve the second problem; (3) dynamic update of the learning model, including incremental feature selection in dynamic feature space and dynamic update of the threshold for classification, which is mainly used to handle the third and fourth problems.

The remainder of this paper is structured as follows. Firstly, we give a series of definitions which closely correlate with the subject of this paper. Secondly, we describe the system framework and working process of ChangeChecker. Then, we discuss the evaluation of ChangeChecker on a large open source project and give a demonstration in a real application scenario. Finally, we draw the conclusion of this paper.

II. DEFINITIONS

In this section, we present several necessary definitions and formal representations which closely correlate with the subject of this paper.

A. Indication Labeling

Given a sequence of source code changes $C=(c_1,c_2,...,c_t)$, the label of a change $c_i(i=1,2,...,t)$ is defined as:

$$l(c_i)\in\{Clean, Buggy\} \quad (1)$$

where a change c_i denotes a source code file commit, *Clean* denotes that there is no defect in change c_i, and *Buggy* denotes that there is at least one defect in change c_i.

B. Features

In this paper, there are mainly five aspects considered as constructing features of source code changes, namely *where* and *when* it happens, *what* happens, *who* manipulates it, and *why* it happens. Each aspect corresponds to a feature group. Formally, we have:

$$F_i=F_i^{where}\cup F_i^{what}\cup F_i^{when}\cup F_i^{who}\cup F_i^{why}(i=1,2,...,t) \quad (2)$$

where F_i denotes the available feature set at time step i, and F_i^{where}, F_i^{what}, F_i^{when}, F_i^{who}, and F_i^{why} denote the feature groups from the five aspects.

1) ***Feature Group Where*****:** In this paper, *where* doesn't indicate the exact location of a source code change but the context of it. The characteristics of source code files that have been touched reflect context information of the source code change. It is well understood that if it is difficult for a source code file to be understood thoroughly, the risk of changing it will be high. Hence, some metrics reflecting the complexity and activity of the touched file are believed to be related to the difficulty of understanding the file and used as features.

a) ***Halstead Metrics*****:** Halstead Metrics were proposed by Maurice Halstead, who argued that the harder the code to read, the more defect prone the modules are [3]. Halstead Metrics are measures of lexical complexity. There are four basic Halstead Metrics (i.e. total number of operators, total number of operands, unique number of operators and unique number of operands) and two derived Halstead Metrics (i.e. volume and programming effort) considered .

b) ***McCabe Metric*****:** Introduced by Thomas McCabe, the idea behind McCabe Metrics is to capture the structural complexity level of a code [4]. The assumption is that it is more likely for the number of defects to increase as the source code gets more complex. Cyclomatic complexity, which is the most famous metric in McCabe Metrics, is considered in this paper.

c) ***Graph Metrics*****:** As indicated by Zimmermann [5], dependencies between software entities are crucial for their successful operation.We assume that the dependency between source code files may correlate with the injection of defects. In this study, we construct a weighted multiple type dependency graph $G=\langle V,E\rangle$, where V denotes the set of nodes (i.e. source code files), and E denotes the set of edges (i.e. dependencies between two files). $\forall e_i\in E(i=1,2,...,M)$, $t(e_i)$ and $w(e_i)$ denote the type and the weight of the edge respectively. There are four types of dependencies considered, namely method invocation, field access, inheritance, and implementation. The weight indicates the amount of dependencies. $\forall v_j\in V(j=1,2,...,N)$, $S_{v_j}=\{e_{s_1},e_{s_2},...,e_{s_m}\}$ denotes the set of edges that take v_j as the start point and $T_{v_j}=\{e_{t_1},e_{t_2},...,e_{t_n}\}$ denotes the set of edges that take v_j as the end point. Formally, the two Graph Metrics weighted out-degree and weighted in-degree are represanted as follows:

$$D^+(v_j)=\sum_{k=1}^{m}w(e_{s_k})\quad D^-(v_j)=\sum_{l=1}^{n}w(e_{t_l}) \quad (3)$$

d) ***History Metrics*****:** Recently, researchers have shown the usefulness of collecting History Metrics from software repositories for defect prediction models [6]. It is assumed that if a source code file has been changed or fixed many times by many developers in the history, the risk of changing it in the future will be high. Hence, there are totally two History Metrics considered in this paper, namely the number of revisions in the last year and the number of fixes in the last year.

e) ***Text Information*****:** Text data contains plenty of semantic function information about source code, which may relate to the injection of defects. Terms in file text and its directory hierarchy are considered as features in this paper.

2) ***Feature Group What*****:** The source code snippets, which have been added, deleted in a source code change, reflect the content about this change. The content contains rich information about what has been done in the change. We take

terms in these snippets as features in this paper. In addition, the number of lines which have been deleted and added in a change is also considered to describe the size of the change.

3) ***Feature Group When*****:** The time when the change occurs can usually capture a developer's habit and work cycle (e.g. Some developers always introduce more defects after midnight) [7].In this paper, we take the time of day and the day of week as two primary features in feature group *when*.

4) ***Feature Group Who*****:** Since defects are introduced into the software by developers, the characteristics of developers should be considered. In this paper, we use login name as a feature to distinguish different developers.

5) ***Feature Group Why*****:** Mocus [8] has pointed out that the intent of the software change (e.g. defect fixing) is related to the injection of defects. Since commit message contains rich information about the change intention, terms in it are considered as features in this paper. Meanwhile, the length of the message, which is used to quantify the information contained in the commit message, is also taken as a feature based on the hypothesis that the more complex the intent of a change is, the more defect prone the change is.

These feature groups almost cover all aspects of a source code change, and are considered to be closely related to the injection of defects.

C. Learning Model

In this paper, the learning model that ChangeChecker based on will be updated with time. At each time step t, it can be denoted as a five tuple:

$$L_t = \langle ST_t, CP_t, TH_t, FS_t, CL_t \rangle \quad (4)$$

where $ST_t = \{st_t^1, st_t^2, ...st_t^n\}$ denotes a collection of statistics tables, n indicates the number of available features, and the table $st_t^i (i = 1,2,...,n)$ holds the number of appearances of the value of feature i for different classes; $CP_t = \{cp_t^1, cp_t^2, ..., cp_t^q\}$ denotes a collection of cut points set, q indicates the number of quantitive features, and $cp_t^j (j = 1,2,...,q)$ denotes cut points set used to discretize the quantitive feature j; TH_t denotes the value used to determine the threshold of classification according to calculated probabilities; FS_t denotes the most predictive features $(f_{s_1}, f_{s_2}, ..., f_{s_m})$ selected from F_t based on ST_t and an incremental feature selection method; and CL_t denotes the probabilities that the current instance belongs to each class, which is obtained by an incremental classifier that calculates the probabilities using ST_t and the selected features FS_t. In this paper, the Naïve Bayes (NB) is chosen as the incremental classifier because it is simple, effective, efficient and robust. Furthermore, the Information Gain (IG) algorithm is taken as the feature selection method because it is inherently incremental and compatible with NB.

D. Learning Task

We use an incremental learning framework. At every time step t, we have historical change sequence with labels, i.e., $C = (c_1, c_2, ..., c_t)$. When a new code change c_{t+1} arrives, the task is to predict the label of c_{t+1}. To this end, we build the learning function L_t based on the feature set $F_t = (f_1, f_2, ..., f_n)$. We apply L_t to predict the label of c_{t+1}. Formally, we have:

$$L_t(c_{t+1} \mid f_1, f_2, ..., f_n) \rightarrow l(c_{t+1}) \quad (5)$$

III. System Framework

The tool based on an incremental learning framework works as follows. (1) Feature collection and transformation. As source code changes constantly arrive with the passing of time, their features should be collected and transformed incrementally; (2) Prediction. ChangeChecker predicts the newly arrived source code change with the current learning model; (3) Update. ChangeChecker collects the user feedback to update the learning model continually, including statistics tables updating, cut points sets updating, and classification threshold updating. Fig. 1 shows the working process of ChangeChecker.

A. Feature Collection and Transformation

This step can be further divided into three parts, namely raw data collection, feature extraction, and transformation of quantitive features.

1) ***Raw Data Collection***: Raw data of source code changes, which is generated during the software development process, should be collected from software history repositories. Here raw data indicates revisions of source code files, commit logs and source code snippets which have been added and deleted in source code changes. This task is mainly implemented by invoking several commands of the version control system.

2) ***Feature Extraction***: After collecting the raw data, ChangeChecker further extracts the features mentioned in above section. The techniques used to extract these features are different since there are various data formats in software history repositories.

Some features that describe the complexity of the internal structure of source files (i.e. Halstead Metrics and McCabe Metric) should be extracted using static analysis technique.

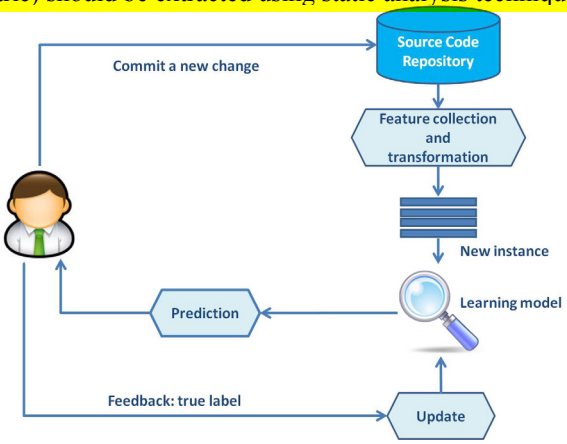

Fig. 1. Incremental learning framework

Features that used to describe the relationships between the current source code file and other files in the project (i.e. weighted out-degree and weighted in-degree) are extracted with incremental building method of software dependency graph proposed in this paper. In order to obtain graph metrics for each revision of the touched file, the evolutionary series of the software dependency graph should be extracted incrementally. As each change arrives, ChangeCheck doesn't rebuild the whole graph, but edits it based on the current change. This editing operation is non-trivial, which should be implemented by converting the source code file change into dependency graph change (i.e. the added nodes and edges, the removed nodes and edges, and the changed nodes and edges). In the converting process, current and previous version of the touched file are compared with each other based on abstract syntax trees to identify modified software entities and their relationships defined in this file. Because the qualified names of some program entities declared in other source files cannot be acquired by parsing each source file alone, the symbol table should be scanned for two times.

Some features can be acquired directly from the version control system (e.g. commit author) or just by simple processing (e.g. time of day and day of week) and computing (e.g. History Metrics).

Text data involved in this paper contains plenty of information about source code changes. Most of the features are extracted from it. ChangeChecker use the BOW and its metamorphosis to generate features from text data, which draws lessons from the work of Kim [9]. For the commit message, which can be considered as common text, features are extracted by taking BOW approach directly. Whereas, for source code file and added/deleted code snippets, in addition to common terms, several important operators (e.g. =, &&, !) should not be ignored. Hence a metamorphosis of BOW called BOW+ [9] is used. Furthermore, the directory and file name are transformed into terms with another metamorphosis of BOW called BOW++ [9], which splits text using slashes and capitals.

*3) **Feature Transformation**:* In order to conform to the data format required by the statistic table, some quantitive features (e.g. length of commit message, the complexity metrics, etc) are transformed into qualitative features. Given by the stream data characteristics of the source code changes, an incremental flexible frequency discretization method IFFD has been adopted [10]. Every time a new change arrives, ChangeChecker will transform its quantitive features into qualitative features based on the current sets of cut points. According to the value distributions of features, the sets of cut points are constantly updated with time.

B. Prediction

The primary purpose of ChangeChecker is to help developers predict whether the incoming source code changes have defects or not. On the arrival of a new change, the tool will make predictions based on the most powerful features and classification threshold at present. When the developer receives the result, he can check the new change to determine if the prediction is true or not, and then give feedbacks to ChangeChecker.

C. Update

When the feedback from developers has been received, the new change is considered as a new training instance used to update the current state of the learning model. The update operation can be divided into three parts, statistics tables updating, cut points sets updating, and classification threshold updating. The statistics tables are updated according to the values of the new change's features and its class label. If the new change has any new features which have not appeared in previous changes before, new statistics tables corresponding these features will be created and added into the collection of statistics tables. The incremental flexible frequency discretization method (IFFD) used in ChangeChecker sets the interval frequency of each feature to be a range instead of a constant. The sets of cut points are updated if the addition of new source code change lead to the frequencies of some features' intervals reach upper limit. The threshold is updated according to the prediction result and the true label of the new change. At the end of update, the feature evaluation metrics are then re-calculated and ranked with the Information Gain method. The whole update process is elaborated in Fig. 2 and Algorithm 1.

IV. EVALUATION

To evaluate ChangeChecker, we have conducted a set of experiments on 5000 sequential code changes from Eclipse JDT data. The result shows the effectiveness of the proposed method. Compared to non-incremental learning model, who has undergone a significant decline as time and got recall lower than 20 percent after 2400 changes, our incremental learning framework with dynamic feature space significantly performs better in dealing with concept drift, and its precision and recall both keep stable at around 70 percent.

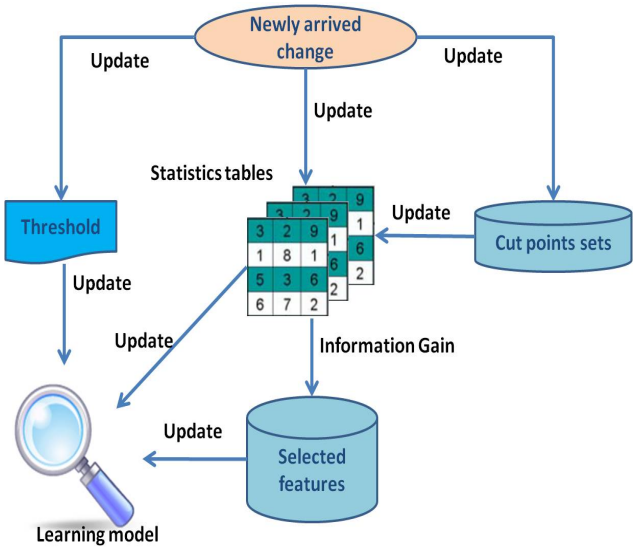

Fig. 2. The update process

Algorithm 1 Update

Input:
l: the label of newly arrived change c_t
l_p: the prediction label of change c_t
pro: probability of c_t belonging to *Buggy*
F_c: features of change c_t
Labelset: the set of labels
QF: the set of quantitive features
F_t: the set of code changes' features
L_t: learning model < ST_t, CP_t, TH_t, FS_t, CL_t >
Output:
F_{t+1}: the set of code changes' features
L_{t+1}: < ST_{t+1}, CP_{t+1}, TH_{t+1}, FS_{t+1}, CL_{t+1}>
1: $CP_{t+1} \leftarrow IFFD(F_c, CP_t, QF)$
2: $TH_{t+1} \leftarrow DynamicThreshold(l, l_p, pro, TH_t)$
3: **for** each feature $f \in F_c$ **do**
4: **if** feature $f \notin F_t$ **then**
5: Add f to set F_t
6: **for** each label $l_e \in Labelset$ **do**
7: **for** each *value* of feature f **do**
8: $ST_t[f][value][l_e] \leftarrow 0$
9: **end for**
10: **end for**
11: **end if**
12: **end for**
13: **for** each feature $f \in F_t$ **do**
14: **if** feature $f \in F_c$ **then**
15: *value* ← the value of change c_t about feature f
16: $ST_t[f][value][l] \leftarrow ST_t[f][value][l]+1$
17: **else**
18: $ST_t[f][0][l] \leftarrow ST_t[f][0][l]+1$
19: **end if**
20: **end for**
21: $F_{t+1} \leftarrow F_t$
22: $ST_{t+1} \leftarrow ST_t$
23: F_{sort} ← sort F_{t+1} based on Information Gain and ST_{t+1}
24: FS_{t+1} ← select subset of F_{t+1} based on F_{sort}
25: CL_{t+1} ← calculate the classification
26: $L_{t+1} \leftarrow$ < ST_{t+1}, CP_{t+1}, TH_{t+1}, FS_{t+1}, CL_{t+1}>

Algorithm 2 IFFD

Input:
F_c: features of change c_t
CP_t: collection of cut points sets
QF: the set of quantitive features
Output:
CP_{t+1}
1: **for** each feature $f \in F_c$ **do**
2: **if** feature $f \in QF$ **then**
3: *value* ← the value of change c_t about feature f
4: *interval* ← get the interval of *value* based on $CP_t[f]$
5: insert *value* into *interval*
6: *freq* ← get the number of values in the *interval*
7: **if** *freq*>upper limit of the interval frequency **then**
8: *newpoint* ← split *interval*
9: add *newpoint* to $CP_t[f]$
10: **end if**
11: **end if**
12: **end for**
13: $CP_{t+1} \leftarrow CP_t$

Algorithm 3 DynamicThreshold

Input:
l: the label of newly arrived change c_t
l_p: the prediction label of change c_t
pro: probability of c_t belonging to *Buggy*
TH_t: threshold of classification
Output:
TH_{t+1}
1: **if** $l \neq l_p$ **then**
2: **if** l = *Clean* and *pro*<1 **then**
3: $TH_t \leftarrow pro$
4: **end if**
5: **if** l = *Buggy* and *pro*>0 **then**
6: $TH_t \leftarrow pro$
7: **end if**
8: **end if**
9: $TH_{t+1} \leftarrow TH_t$

V. DEMONSTRATION

We showcase the ChangeChecker tool using a code evolution parser called "javabear" as the application example, which is being developed by 12 PHD and master students in our lab. There have been 5000 file commits in the version control systems. Fig. 3, 4 and 5 show the user interface of the ChangeChecker. When a developer wants to use the tool ChangeChecker, he should firstly select the "initialize" option and start the initializing process. When he has completed several file modifications and commit them into the version control system, he could verify his commit by selecting the "verify" option. When the verifying process has been finished, the system will list several risk files and their risk area, which might have defects introduced in the last commits. The developer will check and test these risk files and then give feedback to the system. Finally, updating process is started.

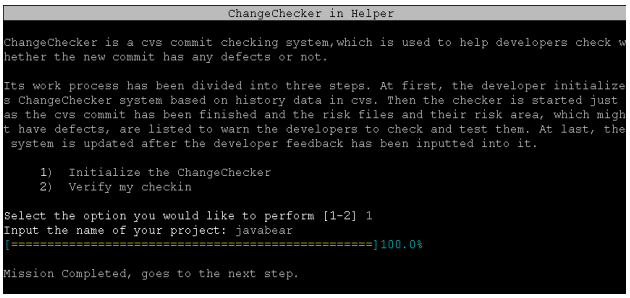

Fig. 3. The initialization of ChangeChecker

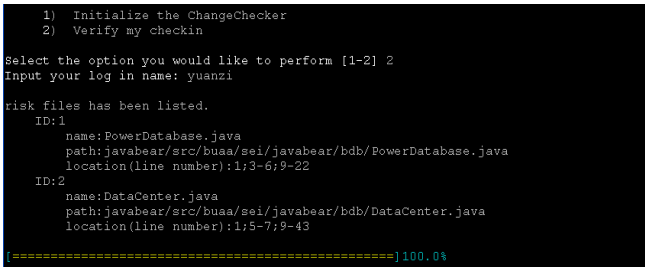

Fig. 4. The Prediction of Risk Files

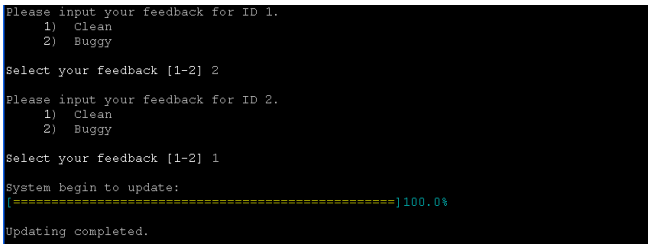

Fig. 5. User Feedback and Update

VI. CONCLUSION

This paper presents a tool called ChangeChecker that is capable of predicting whether the current source code change has any defects during the software development process. The tool concerns about features of source code changes from five different dimensions and is designed based on an incremental learning framework (incremental feature collection and transformation, real-time prediction, and dynamic update of learning model). After evaluation and trial in the real application scenario, it shows that the tool overcomes the concept drift problem, and ensures the high efficiency and good stability of defect prediction.

REFERENCES

[1] E. Giger, M. Pinzger, and H. Gall, "Comparing fine-grained source code changes and code churn for bug prediction", in Proc. MSR, 2011, pp.83-92.

[2] Q. Song, M.J. Shepperd, M. Cartwright, and C. Mair, "Software Defect Association Mining and Defect Correction Effort Prediction", IEEE Transactions on Software Engineering, vol. 32, pp.69-82, 2006.

[3] M. H. Halstead, "Elements of Software Science. Elsevier", 1977.

[4] T.J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, vol. 4, pp.308-320, 1976.

[5] T. Zimmermann and N. Nagappan, "Predicting Subsystem Failures using Dependency Graph Complexities", in Proc. ISSRE, 2007, pp.227-236.

[6] T.L. Graves, A.F. Karr, J.S. Marron, and H.P. Siy, "Predicting Fault Incidence Using Software Change History", IEEE Transactions on Software Engineering, vol. 26, pp.653-661, 2000.

[7] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess", in Proc. MSR, 2011, pp.153-162.

[8] A. Mockus and D. M. Weiss, "Predicting risk of software changes", Bell Labs Technical Journal, pp. 169–180, 2000.

[9] S. Kim, E.J.W. Jr., and Y. Zhang, "Classifying Software Changes: Clean or Buggy?", IEEE Transactions on Software Engineering, vol. 34, pp.181-196, 2008.

[10] J. Lu, Y. Yang, and G. I. Webb, "Incremental discretization for naive-bayes classifier", in ADMA'06, 2006, pp. 223–238.