Personalized Defect Prediction

Tian Jiang
University of Waterloo
Waterloo, ON, Canada
t2jiang@uwaterloo.ca

Lin Tan
University of Waterloo
Waterloo, ON, Canada
lintan@uwaterloo.ca

Sunghun Kim
Hong Kong University of Science and Technology
Hong Kong, China
hunkim@cse.ust.hk

***Abstract*—Many defect prediction techniques have been proposed. While they often take the author of the code into consideration, none of these techniques build a separate prediction model for each developer. Different developers have different coding styles, commit frequencies, and experience levels, causing different defect patterns. When the defects of different developers are combined, such differences are obscured, hurting prediction performance.**

This paper proposes *personalized defect prediction*—building a separate prediction model for each developer to predict software defects. As a proof of concept, we apply our personalized defect prediction to classify defects at the file change level. We evaluate our personalized change classification technique on six large software projects written in C and Java—the Linux kernel, PostgreSQL, Xorg, Eclipse, Lucene and Jackrabbit. Our personalized approach can discover up to 155 more bugs than the traditional change classification (210 versus 55) if developers inspect the top 20% lines of code that are predicted buggy. In addition, our approach improves the F1-score by 0.01–0.06 compared to the traditional change classification.

***Index Terms*—Change classification; machine learning; personalized defect prediction; software reliability**

I. Introduction

Academia and industry expend much effort to predict software defects [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. These prior studies have made significant advances in defect prediction using features such as code complexity, code locations, the amount of in-house testing, historical data, and socio-technical networks.

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

Different developers have different coding styles, commit frequencies, and experience levels, all of which cause different defect patterns [11]. For example, based on our inspection of the Linux kernel's mainline repository from 2005 to 2010, 48.0% of one developer's changes related to `for` loops are buggy while the percentage is only 13.3% for another developer. When the defects of different developers are combined, such differences are obscured, hurting the prediction performance. Therefore, it is desirable to build personalized defect prediction models. Analogously, search engines such as Google use personalized search to capture the different search patterns to provide improved search experience [12].

In this paper, we propose *personalized defect prediction*—building separate prediction models for individual developers to predict software defects. As a proof of concept, we apply our personalized defect prediction idea to change classification [1], which we call ***personalized change classification (PCC)***. Change classification predicts defects at the change level. A *change* is the lines modified in one file of a software version control system commit. In the rest of this paper, we refer to the approach that builds a single change classification model for all developers as the ***traditional change classification (CC)*** approach. In addition, we propose ***PCC+*** to combine PCC and CC to further improve change classification. Many classifiers, such as ADTree [13], provide a confidence measure for each prediction decision. PCC+ picks the prediction from the model with the highest confidence for each change.

Similar to PCC, a recent study by Bettenburg et al. [7] breaks data into clusters and builds separate prediction models for different clusters. That work applies *Multivariate Adaptive Regression Splines (MARS)* [14] to defect prediction, which is a global model approach that takes local considerations into account. In contrast to our approach that groups data by developers, MARS groups data to minimize the fitting error. Bettenburg et al. [7] shows that MARS outperforms the traditional global model approach of building a single classification model from the entire data set. We compare our personalized classification models against MARS models in addition to comparing our personalized classification models against the traditional global models.

To evaluate the proposed approaches, we use two widely used metrics: *Cost Effectiveness* [15], [16], [17] and *F1-score* [1], [15]. Rahman et al. [15] have pointed out that cost effectiveness is a suitable measure for evaluating the performance of defect prediction in cost sensitive scenarios. The cost effectiveness evaluates prediction performance given the same cost, which is typically measured by the lines of code (LOC) to inspect. For example, when a team can afford to inspect only 20% lines of code before a deadline, it is crucial to inspect the 20% that can help the developers discover the most number of bugs. In this paper, we use the same cost effectiveness measure from Rahman et al. [15] with a small variation: the number of bugs instead of the area under the curve. Our cost effectiveness measure is the number of bugs that can be discovered by inspecting 20% LOC (***NofB20***). Both Rahman's and our measures use the

978-1-4799-0215-6/13 © 2013 IEEE

ASE 2013, Palo Alto, USA

same cost effectiveness graph. Rahman's measure considers the area under the curve for the percentage of LOC from 0 to 20%. While Rahman's measure can ensure a high average between 0 and 20%, our measure is much easier to interpret. For example, if PCC improves NofB20 by 100 compared to CC, it means that PCC can help the developers identify 100 more bugs by inspecting the top 20% LOC identified by PCC instead of the top 20% LOC identified by CC. We also present cost effectiveness graphs so that developers can view the number of bugs that can be discovered by inspecting percentages of LOC other than 20%. In addition, we evaluate the approaches on the standard F1-score, which is also widely used in defect prediction [1], [15]. F1-score is an appropriate performance measure when there are enough resources to inspect all predicted buggy changes, e.g., code inspection. If a change is predicted buggy, the developers can put more testing and verification effort into this change. A higher F1-score can help capture more bugs and reducing the time wasted on inspecting clean changes.

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

II. Change Classification (CC)

This section gives a brief overview of the traditional change classification (CC) and our adaptation. The traditional change classification has the following steps [1]:

1) Label each change *clean* or *buggy* by mining the project's revision history [18], [19] (Section II-A). Clean means the code is correct, while buggy means the code contains bugs.
2) Extract features, such as bag-of-words, from the changes (Section II-B). To improve classification performance, we add a new type of features—*characteristic vectors*—to the traditional change classification for the first time. Characteristic vectors can capture some more information about the syntactic structures of the change compared to bag-of-words.
3) Use a classification algorithm to build a model from the labelled changes based on the extracted features (Section II-C). Since MARS [14] has been shown as a superior classification algorithm [7], we use it as another baseline.
4) Predict new changes as buggy or clean using the model.

A. Buggy Change Labelling

We adopt the same definition of *change* as prior work [1]. Compared to defect prediction at the other levels, e.g., the file level, defect prediction at the change level is more precise. For a predicted buggy change, the developer needs to examine only one change instead of the entire file to address the defect.

To label each change as buggy or clean, we follow the method used by previous works [18], [19]. We first identify *bug-fixing* changes—changes that fix bugs–by searching the commit logs for the word "fix". The lines that the bug-fixing changes modified are assumed to be the location of a bug. This approach has a precision of 75–88% in identifying bug-fixing commits in our data set (Section VIII).

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

In addition to the method described above, we pick two projects from the work of Herzig et al. [20], who manually verified the bug reports to distinguish real bugs from feature enhancements. If a bug report fixes a real bug, its associated changes are bug-fixing changes. These two projects have high quality commit messages that contain bug report IDs. Instead of searching for the word "fix", we search for the bug IDs of the real bugs in the commit messages while labeling bug-fixing changes [19]. This approach has a precision of 91–95% in identifying bug-fixing commits in these two projects (Section VIII). We then use the annotate functionality to label bug-introducing changes.

B. Feature Extraction

Features are attributes extracted from a commit which describe the characteristics of the source code, such as LOC. We use three categories of features: 1) characteristic vectors, 2) bag-of-words, and 3) metadata.

1) Characteristic Vector: Inspired by the Deckard tool [21], we use characteristic vectors as features. Characteristic vectors represent the syntactic structure by counting the numbers of each node type in the Abstract Syntax Tree (AST). Bag-of-words (see Section II-B2) and characteristic vectors have different abstraction levels. Although bag-of-words can capture keywords, such as `if` and `while`, it cannot capture abstract syntactic structures, such as the number of statements.

Suppose we are using `if`, `for`, and `while` node types for characteristic vectors. The characteristic vector of the code shown in Figure 1 is (1, 2, 0). After getting the characteristic vectors for the file before the change and the file after the change, we subtract the two characteristic vectors to obtain the difference. For example, for a change that removes one `for` loop, the difference is (0, -1, 0).

For each change, we apply Deckard [21] to automatically generate two characteristic vectors: one for the source code file before the change and one for the source code file after the change. We use the difference between the two characteristic

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

Fig. 1. The characteristic vector for the above code segment contains one `if` statement, two `for` loops and zero `while` loops.

vectors and the characteristic vector of the file after the change as two sets of features.

2) Bag-of-Words: Defects that are caused by calling a wrong function, such as `malloc` instead of `calloc`, cannot be represented by characteristic vectors, because characteristic vectors ignore identifier names. To address this issue, we add the bag-of-words (BOW) [22] vectors as features. It converts a string to a word vector of individual words, where each entry in the vector is the corresponding occurrence of each word.

We use a Weka [23] filter with the Snowball [24] stemmer to convert text strings to word vectors. We process both the commit message and the source code to obtain the word vector for each change.

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

The scope of this paper is not to find the best feature combination; instead it compares personalized prediction models against traditional prediction models, using the same features for both models.

C. Classification and MARS

Given the labels (buggy or clean) and the features extracted from the source code files, machine learning algorithms can learn models and predict new buggy changes. We use off-the-shelf machine learning algorithms from Weka [23].

Since PCC and CC are two different approaches to organize data sets for defect prediction, and neither is tied to any specific classification algorithm, we compare PCC and CC using three widely used [6], [16], [25], [26], [27] classification algorithms: Alternating Decision Tree (ADTree) [13], Naive Bayes [28] and Logistic Regression [29]. This paper does not intend to find the best-fitting classifiers or models, but to compare the PCC models against the CC models using the same classification algorithms.

In addition to CC, we use Multivariate Adaptive Regression Splines (MARS) as another baseline for the following reasons. First, similar to our personalized defect prediction approach, MARS builds separate models for different groups of data to improve the classification performance. Second, MARS performs better than both global models and local models for defect prediction [7].

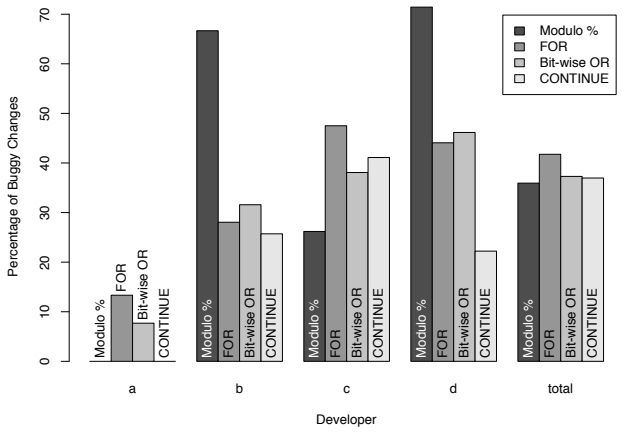


Fig. 2. Buggy rates (percentages of buggy changes) of different syntactic structures of four Linux kernel developers. The syntactic structures are modulo operators (`%`), `for` loops, bit-wise `or` operators (`|`), and `continue` statements. Developers have different buggy change patterns, which cannot be observed if we combine different developers' changes ("total").

We use the MARS implementation in the off-the-shelf machine learning framework Orange [30]. The underlying MARS implementation is the same as used by Bettenburg et al. [7].

III. Personalized Change Classification (PCC)

Different developers have different experience levels, different coding styles, and different commit patterns, resulting in different buggy change patterns [11]. Figure 2 shows differences in buggy change patterns of four prolific Linux kernel developers in the mainline repository from 2005 to 2010. Specifically, it shows the buggy rates (percentage of buggy changes) of four syntactic structures for each developer. The syntactic structures are modulo operators (`%`), `for` loops, bit-wise `or` operators (`|`), and `continue` statements. The x-axis presents different developers and the y-axis indicates the buggy rate for each syntactic structure. The last group "total" shows the sum of all four developers' changes. Each bar shows the ratio of a developer's buggy changes regarding one syntactic structure to the developer's changes regarding the same syntactic structure. For example, 48% of developer c's 320 changes adding or removing `for` loops are buggy.

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

Given such differences in developers' buggy changes, a prediction model built from a developer's changes alone can be more suitable for predicting the changes from the same

developer. Therefore, we build a separate prediction model for each developer. PCC follows exactly the same steps as CC except that PCC groups changes by developers and builds a separate model for each developer. Given a new unlabelled change, the final model first checks the author of the change, and then uses this developer's model to predict this change.

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

Compared to MARS, our personalized change prediction approach is different in two aspects: (1) while MARS partitions data to minimize fitting errors, our PCC approach chooses to partition data by developers, based on our observation that different developers have different development behaviors and patterns; and (2) while a MARS model is a global model that takes local considerations into account, our PCC model is a local model, because the individual model for each developer is learned from a subset of the data. In summary, our PCC approach is a specialized local model approach that utilizes domain specific knowledge.

IV. PCC+: Combining CC and PCC

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

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

Secondly, we automatically pick the prediction with the highest confidence among CC, PCC, and weighted PCC. We refer to this approach as *PCC+*. Many classification algorithms, such as ADTree [13], can provide a *confidence* measure for each prediction. For a prediction, the higher its confidence measure is, the more confident the model is about the prediction. Figure 3 shows the flow diagram of PCC+. For PCC+, we predict each change using these models (CC, PCC and weighted PCC) independently, and pick the prediction with the highest confidence for each change as the final prediction. If two different predictions tie, we favor the predictions in this order: PCC, weighted PCC and CC. An alternative approach to PCC+ is majority voting. We choose the confidence-based approach because majority voting is likely to yield an incorrect prediction when the majority is not confident.

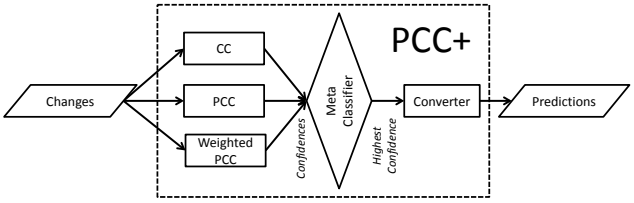


Fig. 3. The flow diagram of PCC+. First, CC, PCC and weighted PCC predict the changes and pass the confidences to the meta classifier. Then, the meta-classifier picks the highest confidence. Last, the converter converts the highest confidence to a prediction (buggy or clean).

V. Experimental Setup

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

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

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

RQ2. Does PCC improves CC on other experimental setups?

First, we investigate if the performance improvement of PCC over CC is generalizable to other classification algorithms. We evaluate PCC and CC on three classification algorithms. We find that **PCC significantly outperforms CC with all evaluated classification algorithms** (Section VI-C).

Second, we investigate the effect of the number of training instances. Since each PCC model learns from a training set that is smaller than that of CC, PCC may not yield a good performance with few changes per developer in smaller projects while CC may still perform well. We investigate how many training instances are enough for PCC to yield a better performance than CC. **In general, PCC models learned from 80 training instances or more per developer can yield a better performance than CC** (Section VI-C).

A. Test Subjects

We choose six open-source projects: the Linux kernel, PostgreSQL, Xorg, Eclipse, Lucene and Jackrabbit. These projects have enough change history to build and evaluate PCC, and they are commonly used in the literature [1], [6], [18], [20], [31]. For Lucene and Jackrabbit, we use manually verified bug reports from Herzig et al. [20] for labeling bug-fixing changes, and the keyword searching approach [19] for the others.

Table I shows detailed project information. The lines of code (LOC) and the number of changes in Table I include only

TABLE I
EVALUATED PROJECTS. THE NUMBERS IN THIS TABLE INCLUDE ONLY C/JAVA CODE. [a]XSERVER SUBREPOSITORY. [b]ECLIPSE JDT CORE SUBREPOSITORY.

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

TABLE II
THE SETUP OF THE DATA SET, SHOWING START GAPS, START DATES, END DATES AND THE BUGGY RATE IN THE DATA SETS. THE BUGGY RATE IS THE OVERALL BUGGY RATE FOR BOTH PCC AND CC SINCE THEY USE EXACTLY THE SAME DATA SETS.

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

source code (C and Java) files[1] and their changes because we want to focus on classifying source code file changes. They are large and typical open source projects covering operating system, database management system and core applications.

Although these projects are written in C and Java, PCC is not limited to any particular programming language. With appropriate feature extraction, PCC could classify changes in any language [1].

B. Data Set

We use the entire revision histories to label buggy changes as described in Section II-A. After labeling buggy changes, we choose changes from a certain period for the following reasons. First, there are too many changes, and often classification algorithms such as MARS do not scale to a large number of instances. In addition, as shown in previous work [18], [32], [33], the average bug life time varies from one year to three years depending on the project. Therefore, the latest changes are often labelled clean even if they are actually buggy simply because the bugs in these changes have not been discovered and fixed. To ensure that we have all necessary fixes to label the buggy changes, we exclude the changes in the last three years. Third, there is also a concern that the change patterns may not be stable at the beginning of the histories [1], [34]. For this reason, we exclude the changes at the beginning of the histories.

Table II shows start gaps, start dates and the average buggy rates of the data sets. The start and end dates are the dates of the first and last commits in the data sets. The start gaps are three years for most of the projects. Lucene and PostgreSQL are two special cases with different start gaps. Since Lucene has a relatively short history (three years), the start gap is six months, and we do not remove any history at the end. PostgreSQL has fewer than five core developers. The required time span to collect enough changes for a non-core developer is long. To collect enough changes, we choose a two-year start gap for PostgreSQL, and we remove the last year of history.

We organize our data sets to perform a fair comparison between PCC and the baselines, i.e., CC and MARS. There are a few key requirements for the experimental setup: 1) we use a mixed data set for the baselines, but developer-specific data sets for PCC; 2) we combine the training sets of PCC to use as the training sets of CC; and 3) we keep the test data the same between PCC and the baselines. With these goals in mind, we select the ten developers from each project who have the most commits. They are the most prolific developers and can represent the majority of each project. We then pick the same number of changes from each of the developers to prevent any developer's performance from dominating. Furthermore, we use 10-fold cross-validation [35] to reduce the bias on the training set selection. This technique is widely used in the literature [1], [7], [16], [36]. PCC is personalized, so we run cross validation on the changes of one developer at a time. For the baselines, we run cross validation on all changes together.

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

C. Classification and Tuning

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

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

In our experiments, we use four classification algorithms, including MARS. We tune each classification algorithm separately. The implementations of our Logistic Regression and Naive Bayes have no tunable parameters [23], so they are not tuned. ADTree has one parameter, the number of boosting iterations [23], so we use a linear search. First, we divide the linear search space into n equally-sized divisions. Then, we evaluate the parameter at the midpoint of each division and pick the parameter value that produces the best result. At last, we recursively repeat the search in the division of the best parameter value. We stop the recursion when the F1 increment between each iteration is less than 0.01. MARS has several parameters that take arbitrary values, e.g., maximum degree, fast k, and penalty for hinges in the GCV computation [30]. Since each run of MARS can take hours, it is infeasible to explore the search space exhaustively. Instead, we tune one parameter first using the above linear search, fix its value, tune the next parameter, and so on.

D. Measures

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

Cost Effectiveness: Cost effectiveness is a widely used metric for defect prediction [15], [16], [17]. As the name suggests, cost effectiveness aims at maximizing benefits by spending the same amount of cost. In our context, the cost is the amount of code to inspect, and the benefit is the number of bugs that can be discovered. If we inspect all predicted buggy changes, the percentage of bugs that we catch is the recall. In some situation, e.g., under deadline pressure, we cannot inspect all predicted buggy changes. Instead, we can inspect perhaps only 20% of committed LOC. It is desirable to catch as many bugs as possible while minimizing LOC to inspect. In this situation, the cost effectiveness metric is a more appropriate measure.

Although our prediction is binary (i.e., a change is either buggy or clean), there can be multiple bugs in a change. It is more appropriate to consider that we can capture multiple bugs after we inspect a change. We borrow the idea from the previous work [15], [37], but adopt it to the change level. Recall the technique that we use to label buggy changes: we use `git blame` on the bug-fixing changes. One bug-introducing change can be `git blame`d by multiple bug-fixing changes, indicating that one change can contain multiple bugs. Therefore, we use the number of bug-fixing changes that `git blame`s a bug-introducing change as the number of bugs in this bug-introducing change.

To evaluate the cost effectiveness, we rank the changes by the probability of being buggy, similar to previous work [15], [16]. We simulate the inspection process by looking at the changes sorted by their rankings. As we inspect the changes, we accumulate the inspected LOC and the discovered bugs. We plot the cost effectiveness graph with the accumulated inspected LOC on the x-axis and the accumulated discovered bugs on the y-axis. Although not restricted to any percentage, we use the number of bugs captured by inspecting 20% of committed LOC, i.e., ***NofB20***, as a quantitative measure similar to previous work [15]. Since different projects have different numbers of bugs, we also normalize NofB20 as a percentage of the total number of bugs, i.e., ***PofB20***. We use NofB20 and PofB20 as the quantitative metrics of cost effectiveness and use the cost effectiveness graph to show the general trend across different percentages of LOC.

An increase in cost effectiveness can help developers find more bugs in cost sensitive scenarios. For example, PCC improves NofB20 by up to 155 compared to CC. It means that developers can discover up to 155 more bugs using the rankings provided by PCC.

F1: F1 is a standard and widely used measure [1], [5], [6] for classification algorithms, which is the harmonic mean of precision and recall. An increase in F1 suggests an increase in precision and recall. Precision represents among all the predicted buggy changes, how many are truly buggy. An increase in precision can reduce the time that developers spend on inspecting true clean changes. Recall represents among all the actual buggy changes, how many the classifier identifies. An increase in recall can help developers capture more bugs. Precision and recall are calculated from numbers of true positives, false positives, false negatives and true negatives. There is a trade-off between precision and recall. Usually, we can sacrifice one to improve the other, which makes it difficult to compare different prediction models using the precision alone or the recall alone.

E. Statistical Tests

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

VI. Experimental Results

A. PCC Versus CC and MARS

In this section, we compare PCC against CC and MARS. We evaluate the three approaches using the setups introduced in Section V, and measure cost effectiveness, precision, recall, F1, NofB20 and PofB20 as described in Section V-D. In

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

TABLE III
RESULTS OF EVALUATED PROJECTS. THE VALUES IN PARENTHESES SHOW THE F1, NOFB20 AND POFB20 DIFFERENCES AGAINST CC. THE "AVERAGE" ROW CONTAINS THE AVERAGE IMPROVEMENT OF PCC OVER CC ACROSS ALL PROJECTS. STATISTICALLY SIGNIFICANT IMPROVEMENTS ARE BOLDED.

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

addition, we use statistical tests to check if result differences are statistically significant as described in Section V-E.

Table III presents the overall classification performance of CC, MARS and PCC. The values in parentheses show the improvements of F1, NofB20 and PofB20 against CC. The "Average" row contains the arithmetic means of the improvements between PCC and CC across all projects. The statistically significant improvements are bolded. We use ADTree in RQ1 since decision tree is widely used [16], [26].

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

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

The cost effectiveness graph for Lucene is shown in Figure 4. The other projects have similar trends and are not shown due to space constraints. As shown in the figure, PCC and CC diverge when the percentage of LOC is at about 5% and converge around 60%. It means that PCC is better than CC for a wide range of LOC choices other than 20%.

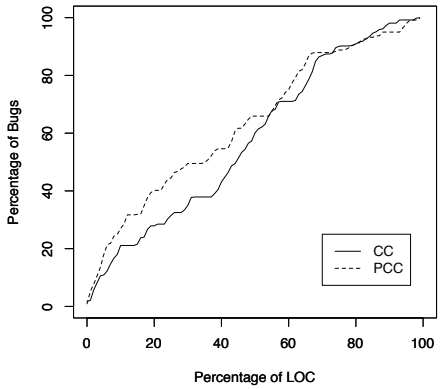


Fig. 4. Cost effectiveness graph for Lucene. It shows the percentage of bugs that can be discovered by inspecting different percentages of LOC. PCC is better than CC for a wide range of LOC choices.

F1: As shown in Table III, PCC improves CC by up to 0.06 on F1. For example, CC's F1 on Eclipse is 0.53, and PCC's F1 on Eclipse is 0.59, which indicates that PCC improves the F1 by 0.06. The statistically significant deltas are bolded. Although the delta on PostgreSQL is negative, the associated p-values indicates that the difference is not statistically significant. It means that we find no evidence that PCC and CC perform differently predicting PostgreSQL in terms of F1. Unlike the other subjects, the PostgreSQL community requires the author field to reflect committers rather than actual authors. Since we do not have the information of actual authors, PCC's advantage may not show, which could affect the performance of PostgreSQL (Section VIII). As shown in the "Average" row, **PCC outperforms CC on average across all test subjects**.

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

We have shown that PCC's F1 is higher than CC's. In addition, PCC has comparable or higher precision than CC for all projects. Some developers may prefer precision while others prefer recall. To address this issue, we can trade recall for precision and vice versa by simply tuning the classification algorithm parameters. Kim et al. [1] showed the trade-off between precision and recall.

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

MARS's local consideration does not explicitly distinguish developers. Our result shows that PCC, a local model which

TABLE IV
F1 AND NOFB20 FOR DIFFERENT CLASSIFIERS. THE DELTA BETWEEN CC AND PCC ARE SHOWN IN PARENTHESES. THE "AVERAGE" ROW CONTAINS THE AVERAGE IMPROVEMENT OF PCC OVER CC ACROSS ALL PROJECTS FOR EACH CLASSIFICATION ALGORITHM. DUE TO LIMITED SPACE, THE P-VALUES ARE NOT SHOWN. INSTEAD, STATISTICALLY SIGNIFICANT DELTAS ARE BOLDED.

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

utilizes domain specific knowledge, outperforms a global model with local consideration.

PCC outperforms both CC and MARS.

B. PCC+

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

PCC+ is a meta-classifier that picks the most confident result among CC, PCC and weighted PCC as described in Section IV. Recall that we use ADTree in RQ1. According to Freund et al. [13], an ADTree-based classifier can provide a confidence measure. For each change, we predict using the three models, i.e., CC, PCC and weighted PCC. PCC+ picks the result that has the highest confidence. In the case of a tie, we favor the prediction in this order: PCC, weighted PCC and CC.

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

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

C. PCC's Improvement on Other Settings

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

Different Classification Algorithms: We deploy three classification algorithms to check if PCC outperforms CC with other classification algorithms. We conduct this experiment using the same data sets in Section VI-A.

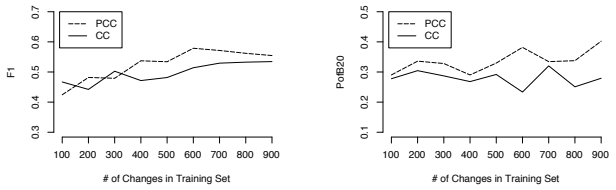
Table IV shows the results of PCC and CC using various classifiers. The columns show the project name, the approach, the CC and PCC results with ADTree classifiers taken from Table III, and the results with other classifiers—Naive Bayes and Logistic Regression. The results demonstrate that PCC outperforms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

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

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

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

Figure 5 shows the relationship between the number of training instances versus F1 and PofB20 for Lucene. For example, the points corresponding to 300 on the x-axis shows the F1 or PofB20 of using 300 training instances (30 per developer). The results show that there is an overall trend of



(a) F1 (b) PofB20

Fig. 5. F1 and PofB20 versus number of training changes for Lucene. X-axis is the number of training changes for CC, and 1/10 of it is the number of training changes per developer for PCC.

F1 and PofB20 increasing with the number of the training instances. Having the same trend, the other projects are not shown due to space constraints.

We can compare PCC and CC results by comparing the corresponding lines in Figure 5. Although PCC outperforms CC in general, the results that show PCC and CC are very close to each other at the beginning, and then diverge. Although not shown due to space constraints, different projects generally diverge at different points before 80 training instances per developer. When there are fewer training instances, the advantage of PCC does not show, because there are not enough training instances to build a classification model, while CC has enough instances since CC has 10 times more training instances than each PCC model. Overall, our data show that it is generally worthwhile to use PCC for better performance when there are 80 or more training instances per developer in the project.

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

VII. Discussion

Interpretation of Results: Although being useful in the build process, developers often ignore predictions that are not explained [38]. The good news is, defect prediction techniques can be improved to provide explanations, e.g., which features lead to a buggy prediction. For example, an ADTree model is a tree that shows step by step how it makes predictions based on the features [13]. Figure 6 shows three ADTree nodes from three developers' trees. This figure shows how the bit-wise `and` operator can affect the decision process in three developers' models. For each change, we find how many bit-wise `and` operators the change adds or removes. For example, "-1" represents removing one bit-wise `and` operator. We compare the modifications against a threshold. "<-0.5" means to check if the change removes more than 0.5 bit-wise `and` operators. In summary, if developer dev2 removes more than 2.5 bit-wise `and` operators in a change, the weight of this change being buggy increases by 0.815. We traverse the tree by visiting the node at each level that matches the change. The change is predicted buggy if the sum of the weights along the path is positive. Developers dev1 and dev2 are more likely to produce buggy changes if they remove bit-wise `and` operators. In contrast, developer dev3 is more likely to produce clean changes if she removes bit-wise `and` operators. These examples confirm that there are subtle differences between developers and the classification algorithms can catch such differences.

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

Fig. 6. These three ADTree nodes are from three developers' trees. The first node means that the weight of a change being buggy increases by 0.815 if developer dev1 removes any bit-wise `and` operator in this change. A change is predicted buggy if the weight is positive.

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

VIII. Threats to Validity

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

Bug data contain noise: Since we follow the traditional change classification techniques [1] to identify bug-fixing changes, our data inevitably include noise as Bird et al. pointed out [39]. For this reason, we carefully select our subjects to have high quality commit logs. In addition, we use two projects that have manually verified bug reports, Lucene and Jackrabbit [20]. For all subjects, we manually check the noise level. We randomly sample 200 bug-fixing commits from each project and manually verify whether they are indeed bug fixes. We find that the precision and recall are reasonable: the precision and recall are 0.87 and 0.73 for the Linux kernel, and 0.86 and 0.71 for PostgreSQL [18]; the precision and recall are 0.75 and 0.64 for Xorg, 0.78 and 0.93 for Eclipse, 0.91 and 0.93 for Lucene, and 0.95 and 0.87 for Jackrabbit. These noise levels should be acceptable [36]. However, this noise may affect our results even though we use the same data sets

for PCC, CC, and MARS. In the future, we can reduce the noise levels through advanced techniques [40], [41].

In addition, the developer information may be inaccurate for PostgreSQL. The PostgreSQL community currently requires that both the author field and the committer fields to reflect the committer [42], so we do not have the actual author information for many commits, which may hurt our results. In the future, we may analyze PostgreSQL's commit messages and mailing lists to extract the author information to potentially improve PCC's performance on PostgreSQL. In addition, it may be interesting to compare author-specific change classification with committer-specific change classification.

Developer and change selection might affect our results: For PCC, we select the ten developers per project with the most commits, and use 100 changes per developer as our data sets (Table II). Note that we used the same data sets for PCC, CC and MARS. We select these top developers because there are too many inactive developers in open source projects who committed only a few changes. Removing these inactive developers from experiments is a common practice in the literature [43], [44], [45], [46]. We select the same number of changes per developer since some developers have too many changes. For example, one developer in PostgreSQL committed about 41% of all changes in the project history. We do not want a large number of changes from a few developers to dominate the classification results. These selections might affect our experimental results; alternative ways of selecting developers and changes remain as our future work.

IX. Related Work

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

Many studies [1], [2], [3], [4], [5], [8], [9], [10], [16], [47] analyze the effects of factors such as code complexity, process metrics, code locations, the amount of in-house testing, historical data, and socio-technical networks on building prediction models and predicting software defects. Kim et al. [1] use support vector machines (SVM) as the classifier with bag of words, complexity metrics and metadata as the features to predict defects. Recently, Shivaji et al. [48] improve the work above by applying feature selection algorithms. Ostrand et al. [11] use negative binomial regression as the classifier, and various metadata and developer-specific metrics as features to predict defects. Rahman et al. [16] compare the effect of code metrics and process metrics. Lumpe et al. [47] investigate the importance of activity-centric static code metrics. These studies advanced the state of the art of defect prediction. However, none of them builds personalized models although some of them consider the developers as an important feature. We notice the importance of the difference between the developers, and improve the change classification performance by building personalized models.

The personalization idea has great success in other fields. For example, Google's personalized search can enhance search results by leveraging the users' search history [12]. Facebook's and MySpace's personalized advertisement deployment can predict user's interest by analyzing the user's profile [49]. We introduce the personalized idea to the change classification field. Our results show that the personalized approach can improve change classification performance.

Bettenburg et al. [7] find that MARS, a global model that has local considerations, is better than both global models and local models. Although MARS groups data intro clusters, it does not group data by developer. A detailed comparison has been presented in Section III.

Matsumoto et al. [50] find that developer-related metrics are good distinguishing factors for defect prediction. Specifically, the modules that are touched by more developers contain more bugs. Rahman and Devanbu [51] find that a developer's experience on one file is more important than the developer's general experience on the project. Posnett et al. [52] introduce focus metrics and find that more focused developers introduce fewer bugs. In the future, we may add these developer related features to improve our defect prediction.

Rahman et al. [15] argue that defect prediction should also consider the cost effectiveness metrics, and show that using F1-score and cost effectiveness can yield different results.

Herzig et al. [20] show that misclassified bug reports have a great impact on defect prediction, and provide manually classified bug reports. Our study follows the lessons learned by reporting cost effectiveness metrics and using the manually classified bug reports in our evaluation.

X. Conclusions

We propose personalized defect prediction and apply it to change classification as a proof of concept. In addition, we propose PCC+ to further improve performance by combining models based on confidence measures. Our empirical evaluation of PCC on six open source projects shows that our personalized change classification outperforms the traditional change classification [1] and MARS [7]. We also find that the advantage of PCC is not bounded to any classification algorithm. In general, PCC outperforms CC when there are more than 80 training instances per developer.

Our personalized idea could be applied to recommendation systems [53], [54], [55] and other types of predictions, such as top crashes [56], quality attributes [57], bug-fixing commits [58], vulnerabilities [59], bug location and number of bugs [37]. Since each developer may have different software development behaviors, personalized systems can effectively learn from a developer and provide useful information for the developer.

Acknowledgments

We thank Tsu-Wei Webber Chen and Mitchell Jameson for their help with the experiments. We also thank William Marshall and the University of Waterloo's statistical counseling service for the help with the statistical analysis. This work is supported by the National Science and Engineering Research Council of Canada and a Google gift grant.

REFERENCES

[1] S. Kim, J. E. James Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *TSE*, vol. 34, pp. 181–196, 2008.

[2] A. Hassan, "Predicting faults using the complexity of code changes," in *ICSE*, 2009, pp. 78–88.

[3] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *FSE*, 2008, pp. 13–23.

[4] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. B. Bener, "Defect prediction from static code features: Current results, limitations, new approaches," *ASE*, 2010.

[5] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *ICSE*, 2008.

[6] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects for Eclipse," in *PROMISE*, 2007.

[7] N. Bettenburg, M. Nagappan, and A. E. Hassan, "Think locally, act globally: Improving defect and effort prediction models," in *MSR*, 2012, pp. 60–69.

[8] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *TSE*, vol. 26, no. 7, pp. 653–661, 2000.

[9] I. Herraiz, J. González-Barahona, G. Robles, and D. Germán, "On the prediction of the evolution of libre software projects," in *ICSM*, 2007, pp. 405–414.

[10] A. Mockus, D. Weiss, and P. Zhang, "Understanding and predicting effort in software projects," in *ICSE*, 2003.

[11] T. J. Ostrand, E. J. Weyuker, R. M. Bell, P. Avenue, and F. Park, "Programmer-based fault prediction," in *PROMISE*, 2010, pp. 1–10.

[12] Google Official Blog, "Personalized search for everyone," http://googleblog.blogspot.ca/2009/12/personalized-search-for-everyone.html, 2009.

[13] Y. Freund and L. Mason, "The alternating decision tree learning algorithm," in *ICML*, 1999, pp. 124–133.

[14] J. H. Friedman, "Multivariate adaptive regression splines," *The Annals of Statistics*, vol. 19, no. 1, pp. 1–67, 1991.

[15] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the "imprecision" of cross-project defect prediction," in *FSE*, 2012.

[16] F. Rahman and P. Devanbu, "How, and Why, process metrics are better," in *ICSE*, 2013.

[17] E. Arisholm, L. C. Briand, and M. Fuglerud, "Data mining techniques for building fault-proneness models in telecom java software," 2007, pp. 215–224.

[18] J. Eyolfson, L. Tan, and P. Lam, "Correlations between bugginess and time-based commit characteristics," *EMSE*, pp. 1–31, 2013.

[19] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do Changes Induce Fixes?" in *MSR*, 2005, pp. 24–28.

[20] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *ICSE*, 2013.

[21] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *ICSE*, 2007, pp. 96–105.

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

[23] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, 2009.

[24] "Snowball," http://snowball.tartarus.org/.

[25] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now? An empirical study of bug characteristics in modern open source software," in *ASID*, October 2006.

[26] G. Bougie, C. Treude, D. Germán, and M. Storey, "A comparative exploration of FreeBSD bug lifetimes," in *MSR*, 2010, pp. 106–109.

[27] 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 *FSE*, 2009, pp. 91–100.

[28] G. H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," in *UAI*. San Mateo: Morgan Kaufmann, 1995, pp. 338–345.

[29] N. Landwehr, M. Hall, and E. Frank, "Logistic model trees," vol. 95, no. 1-2, pp. 161–205, 2005.

[30] T. Curk, J. Demar, Q. Xu, G. Leban, U. Petrovi, I. Bratko, G. Shaulsky, and B. Zupan, "Microarray data mining with visual programming," *Bioinformatics*, vol. 21, pp. 396–398, Feb. 2005.

[31] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *SOSP*, 2001, pp. 57–72.

[32] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *SOSP*, 2001, pp. 73–88.

[33] S. Kim and E. J. Whitehead, Jr., "How long did it take to fix bugs?" in *MSR*, 2006, pp. 173–174.

[34] A. Hassan and R. Holt, "The top ten list: Dynamic fault prediction," in *ICSM*, 2005, pp. 263–272.

[35] A. W. Moore, "Cross-validation," http://www.autonlab.org/tutorials/overfit.html, 2008.

[36] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *ICSE*, New York, NY, USA, 2011, pp. 481–490.

[37] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *TSE*, vol. 31, no. 4, pp. 340–355, Apr. 2005.

[38] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr, "Does bug prediction support human developers? findings from a google case study," in *ICSE*, 2013, pp. 372–381.

[39] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: Bias in bug-fix datasets," in *FSE*, 2009, pp. 121–130.

[40] R. Wu, H. Zhang, S. Kim, and S. Cheung, "Relink: Recovering links between bugs and changes," in *FSE*, 2011, pp. 15–25.

[41] S. Kim, T. Zimmermann, K. Pan, and J. J. Whitehead, "Automatic identification of bug-introducing changes," in *ASE*, 2006, pp. 81–90.

[42] PostgreSQL Community, "Committing with Git – PostgreSQL Wiki," http://wiki.postgresql.org/wiki/Committing_with_Git, 2012.

[43] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *ICSE*, 2006, pp. 361–370.

[44] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *MSR*, 2009, pp. 131–140.

[45] J. Park, M. Lee, J. Kim, S. Hwang, and S. Kim, "CosTriage: A cost-aware triage algorithm for bug reporting systems," in *AAAI*, 2011.

[46] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *FSE*, 2009, pp. 111–120.

[47] M. Lumpe, R. Vasa, T. Menzies, R. Rush, and B. Turhan, "Learning better inspection optimization policies," *IJSEKE*, vol. 22, no. 5, pp. 621–644, 2012.

[48] S. Shivaji, E. J. W. Jr., R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," *TSE*, vol. 39, no. 4, pp. 552–569, 2013.

[49] C. Tucker, "Social Networks, Personalized Advertising, and Privacy Controls," in *WEIS*, 2011.

[50] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *PROMISE*, 2010, pp. 18:1–18:9.

[51] F. Rahman and P. Devanbu, "Ownership, experience and defects: A fine-grained study of authorship," in *ICSE*, 2011, pp. 491–500.

[52] D. Posnett, R. DSouza, P. Devanbu, and V. Filkov, "Dual ecological measures of focus in software development," in *ICSE*, 2013.

[53] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *ICSE*, 2012, pp. 837–847.

[54] K. Muslu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Improving ide recommendations by considering global implications of existing recommendations," in *ICSE*, 2012, pp. 1349–1352.

[55] Y. Ye and G. Fischer, "Supporting reuse by delivering task-relevant and personalized information," in *ICSE*, 2002, pp. 513–523.

[56] D. Kim, X. Wang, S. Kim, A. Zeller, S. Cheung, and S. Park, "Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts," *TSE*, vol. 37, no. 3, pp. 430–447, 2011.

[57] S. S. Kolesnikov, S. Apel, N. Siegmund, S. Sobernig, C. Kästner, and S. Senkaya, "Predicting quality attributes of software product lines using software and network measures and sampling," in *VaMoS*, 2013, p. 6.

[58] Y. Tian, J. Lawall, and D. Lo, "Identifying Linux bug fixing patches," in *ICSE*, 2012, pp. 386–396.

[59] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *EMSE*, vol. 18, no. 1, pp. 25–59, 2013.