Reducing Features to Improve Bug Prediction

Shivkumar Shivaji, E. James Whitehead, Jr., Ram Akella
University of California Santa Cruz
{shiv,ejw,ram}@soe.ucsc.edu

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

***Abstract*—Recently, machine learning classifiers have emerged as a way to predict the existence of a bug in a change made to a source code file. The classifier is first trained on software history data, and then used to predict bugs. Two drawbacks of existing classifier-based bug prediction are potentially insufficient accuracy for practical use, and use of a large number of features. These large numbers of features adversely impact scalability and accuracy of the approach. This paper proposes a feature selection technique applicable to classification-based bug prediction. This technique is applied to predict bugs in software changes, and performance of Naïve Bayes and Support Vector Machine (SVM) classifiers is characterized.**

***Index Terms*—Reliability; Bug prediction; Machine Learning; Feature Selection**

I. Introduction

Classifiers, when trained on historical software project data, can be used to predict the existence of a bug in an individual file-level software change, as demonstrated in prior work by the second and fourth authors [1] (hereafter called Kim et al.), in work by Hata et al. [2], and others. The classifier is first trained on information found in historical changes, and can be used to classify a new change as being either buggy (predicted to have a bug) or clean (predicted to not have a bug). Though these results rank among the best bug prediction algorithms, they are perhaps not strong enough to be used in practice.

We envision a future where software engineers have bug prediction capabilities built into their development environment. Software engineers will receive feedback from a classifier on whether each change they commit is either buggy or clean. In recent work we have created a prototype displaying server-computed bug predictions inside the Eclipse IDE [3]. A bug prediction service must provide highly precise predictions. If engineers are to trust a bug prediction service, it must provide very few "false alarms", changes that are predicted to be buggy but which are really clean. If too many clean changes are falsely predicted to be buggy, developers will lose faith in the bug prediction system.

The prior change classification bug prediction approach used by Kim et al. involves the extraction of "features" (in the machine learning sense, which differ from software features) from the history of changes made to a software project. These features include everything separated by whitespace, in the code added or deleted in a change. This leads to a large number of features, in the thousands, and low tens of thousands. For larger project histories which span thousand revisions or more, this can stretch into hundreds of thousands of features.

The large feature set comes at a cost. The addition of many non-useful features reduces a classifier's accuracy. Additionally, the time required to perform classification increases with the number of features, rising to several seconds per classification for tens of thousands of features, and minutes for large project histories.

A standard approach (in the machine learning literature) for handling large feature sets is to perform a feature selection process to identify that subset of features providing the best classification results. This paper introduces a feature selection process that discards features with lowest gain ratio until optimal classification performance is reached for a given performance measure.

This paper explores the following research questions.

Question 1. Which choices lead to best bug prediction performance using feature selection?
The two variables affecting bug prediction performance that are explored in this paper are: (1) type of classifier (Naïve Bayes, Support Vector Machine), and (2) which metric (recall, F-measure) is optimized for the classification. Results are reported in Section IV-A.

Results for question 1 are reported as averages across all projects in the corpus. However, in practice it is useful to know the range of results across a set of projects. This leads to our second question.

Question 2. Range of bug performance using feature selection. What is the range of performance of the best-performing Bayesian (F-measure optimized) classifier across all projects when using feature selection? (see Section IV-B)

The primary contribution of this paper is the process of using Gain Ratio for feature selection, along with the characterization of bug prediction results achieved when using feature selection. A comparison of this paper's results with those found in related work (see Section V) show that change classification with feature selection outperforms other existing classification-based bug prediction approaches. Furthermore, when using Naïve Bayes (F-measure optimized) buggy precision averages .96, indicating the bug predictions are generally highly precise, thereby avoiding the "false negatives" problem.

In the remainder of the paper, we begin by presenting an overview of the change classification approach for bug prediction, and then detail the new algorithm for feature selection (Section II). Following, we describe the experimental context, including our data set, and specific classifiers (Section III). The stage is now set, and in subsequent sections we explore the research questions described above (Sections IV-A

- IV-B). The paper ends with a summary of related work (Section V), and the conclusion.

II. Change Classification

The primary steps involved in performing change classification on a single project are outlined as follows:

Creating a corpus:

1. File level changes are extracted from the revision history of a project, as stored in its SCM repository (described further in Section II-A).

2. The bug fix changes for each file are identified by examining keywords in SCM change log messages (Section II-A).

3. The bug-introducing and clean changes at the file level are identified by tracing backward in the revision history from bug fix changes (Section II-A).

4. Features are extracted from all changes, both buggy and clean. Features include all terms in the complete source code, the lines modified in each change (delta), and change metadata such as author and change time. Complexity metrics, if available, are used at this step. Details on these feature extraction techniques are presented in Section II-B.

All of the steps until this point are the same as in Kim et al. The following step is the new contribution in this paper.

Feature Selection:

5. Perform a feature selection process that employs Gain Ratio to compute a reduced set of features, as described in Section II-C. For each iteration of feature selection, classifier performance is optimized for a metric (typically F-measure or accuracy). Feature selection is iteratively performed until optimum points are reached. At the end of Step 5, there is a reduced feature set that performs optimally for the chosen classifier metric.

Classification:

6. Using the reduced feature set, a classification model is trained. Although many classification techniques could be employed, this paper focuses on the use of Naïve Bayes and SVM.

7. Once a classifier has been trained, it is ready to use. New changes can now be fed to the classifier, which determines whether a new change is more similar to a buggy change or a clean change.

A. Finding Buggy and Clean Changes

In order to find bug-introducing changes, bug fixes must first be identified by mining change log messages. We use two approaches: searching for keywords in log messages such as “Fixed”, “Bug” [4], or other keywords likely to appear in a bug fix and searching for references to bug reports like “#42233”. This allows us to identify whether an entire code change transaction contains a bug fix. If it does, we then need to identify the specific file change that introduced the bug. For the systems studied in this paper, we manually verified that the identified fix commits were, indeed, bug fixes. For JCP, all bug fixes were identified using a source code to bug tracking system hook. As a result, we did not have to rely on change log messages for JCP.

The bug-introducing change identification algorithm proposed by Śliwerski, Zimmermann, and Zeller (SZZ algorithm) [5] is used in the current paper. After identifying bug fixes, SZZ uses a difference tool to determine what changed in the bug-fixes.

B. Feature Extraction

A file change involves two source code revisions (an old revision and a new revision) and a change delta that records the added code (added delta) and the deleted code (deleted delta) between the two revisions. A file change has associated metadata, including the change log, author, and commit date. Every term in the source code, change delta, and change log texts is used as a feature.

We gather eight features from change metadata: author, commit hour, commit day, cumulative change count, cumulative bug count, length of change log, changed LOC (added delta LOC + deleted delta LOC), and new revision source code LOC.

We compute a range of traditional complexity metrics of the source code by using the Understand C/C++ and Java tools [6].

To generate features from source code, we use a modified version of the bag-of-words approach (BOW) [7], called BOW+, that extracts operators in addition to all terms extracted by BOW, since we believe operators such as !=, ++, and && are important terms in source code. We perform BOW+ extraction on added delta, deleted delta, and new revision source code.

C. Feature Selection Algorithm

The number of features gathered during the feature extraction phase is quite large, ranging from 6,127 for Plone to 41,942 for JCP (Table I). Such large feature sets lead to longer training and prediction times, requiring large amounts of memory to perform classification. A common solution to this problem is the process of feature selection, in which only the subset of features that are most useful for making classification decisions are actually used.

The primary tool used in this paper to determine the most useful features is Gain Ratio based feature selection. Gain Ratio improves upon Information Gain [8], a well known entropy based measure of the amount by which a given feature contributes information to a classification decision.

Gain ratio plays the same role as information gain, but instead provides a normalized measure of a feature’s contribution to a classification decision [8]. We found Gain Ratio to be one of the best performing feature selection techniques on bug prediction data after investigating many others. More details on how the entropy based measure is calculated for Gain Ratio and its inner workings can be found in an introductory data mining book, e.g. [8].

Gain Ratio is used in an iterative process of selecting incrementally smaller sets of features, as detailed in Algorithm 1. The feature selection algorithm begins by cutting the

Algorithm 1 Feature selection algorithm for one project

1) Start with all features, F
2) Compute Gain Ratio over F, and select the top 50% of features with the best Gain Ratio, $F/2$
3) Selected features, $selF = F/2$
4) While $|selF| \geq 0.1\%|F|$, perform steps (a)-(d)
   a) Compute and store buggy and clean precision, recall, accuracy, F-measure and ROC AUC using the a machine learning classifier (e.g., Naïve Bayes or SVM), using 10-fold cross validation
   b) Compute Gain Ratio over $selF$
   c) Identify $removeF$, the 10% of features of $selF$ with the lowest Gain Ratio. These are the least useful features in this iteration.
   d) $selF = selF - removeF$
5) For a given classifier metric (e.g., accuracy, F-measure), determine the best result recorded in step 4.a. The percentage of features that yields the best result is optimal for the given metric.

initial feature set in half, to reduce memory and processing requirements for the remainder of the algorithm. Since optimal feature sets are typically found at under 10% of all features, this reduces algorithm iterations. Projects with large numbers of features can be even more aggressive in the initial feature selection; for PostgreSQL only 25% and for JCP only 12.5% of all features were used for the initial $selF$.

In the iteration stage, each iteration finds those 10% of remaining features that are least useful for classification, and eliminates them (if, instead, we were to reduce by one feature at a time, this step would be similar to backward feature selection [9]). The benefit of using our proposed number of 10% of features at a time is improved algorithm speed while maintaining result quality. So, for example, $selF$ starts at 50% of all features, then is reduced to 45% of all features, then 40.5%, and so on. At each step, change classification bug prediction using $selF$ is then performed over the entire revision history, using 10-fold cross validation to reduce the possibility of over-fitting to the data.

This iteration terminates when there are only a few features remaining. At this point, there is a list of (feature %, classifier evaluation metric) tuples. The final step involves a pass over this list to find the feature % at which a specific classifier evaluation metric achieves its greatest value. The two metrics explored in this paper are accuracy, and buggy F-measure (aggregate of buggy precision and recall), though other metrics could be further investigated. Definitions of accuracy, F-measure, ROC can be found in an introductory machine learning text.

III. Experimental Context

We gathered software revision history for Apache, Columba, Gaim, Gforge, Jedit, Mozilla, Eclipse, Plone, PostgreSQL, Subversion, and a commercial project written in Java (JCP).

TABLE I
Summary of Projects Surveyed

Project	Period	Clean Changes	Buggy Changes	Features
Apache 1.3	10/1996-01/1997	579	121	17,575
Columba	05/2003-09/2003	1,270	530	17,411
Gaim	08/2000-03/2001	742	451	9,281
GForge	01/2003-03/2004	339	334	8,996
JEdit	08/2002-03/2003	626	377	13,879
Mozilla	08/2003-08/2004	395	169	13,648
Eclipse	10/2001-11/2001	592	67	16,192
Plone	07/2002-02/2003	457	112	6,127
PostgreSQL	11/1996-02/1997	853	273	23,247
Subversion	01/2002-03/2002	1,925	288	14,856
JCP	1 year	1,516	403	41,942
Total	N/A	9,294	3,125	183,054

These are all mature open source projects with the exception of JCP. These projects are collectively referred to in this paper as the corpus.

Using the project's CVS (Concurrent Versioning System) or SVN (Subversion) source code repositories, we collected revisions 500-1000 for each project, excepting Jedit, Eclipse, and JCP. For Jedit and Eclipse, revisions 500-750 were collected. For JCP, a year's worth of changes were collected. Table I provides an overview of the projects examined in this research and the duration of each project examined.

IV. Results

The following sections present results obtained when exploring the three research questions.

A. Classifier performance comparison

The two main variables affecting bug prediction performance that are explored in this paper are: (1) type of classifier (Naïve Bayes, Support Vector Machine), and (2) which metric (accuracy, F-measure) the classifier is optimized on. The four permutations of these variables are explored across all 11 projects in the data set. For SVMs, a linear kernel with standard values for slack is used. For each project, feature selection is performed, followed by computation of per-project accuracy, buggy precision, buggy recall, and buggy F-measure. Once all projects are complete, average values across all projects are computed. Results are reported in Table II.

B. Effect of feature selection

In the previous section, aggregate average performance of different classifiers and optimization combinations is compared across all projects. In actual practice, change classification would be trained and employed on a specific project. As a result, it is useful to understand the range of performance

TABLE II
AVERAGE CLASSIFIER PERFORMANCE ON CORPUS

Technique	Features Percentage	Accuracy	Buggy Precision	Buggy Recall	Buggy F-measure
Bayes F-measure	6.83	0.91	0.96	0.67	0.79
SVM F-measure	7.49	0.81	0.82	0.54	0.62
Bayes accuracy	6.92	0.86	0.92	0.53	0.65
SVM accuracy	7.83	0.86	0.96	0.51	0.65

TABLE III
NAÏVE BAYES USING F-MEASURE OPTIMIZED FEATURE SELECTION

Project Name	Features	Accuracy	Buggy Precision	Buggy Recall	Buggy F-measure	Buggy ROC
APACHE	465	0.92	1	0.56	0.72	0.78
COLUMBA	1618	0.9	0.99	0.67	0.8	0.83
ECLIPSE	802	0.98	0.98	0.79	0.88	0.88
GAIM	1065	0.86	0.96	0.65	0.78	0.83
GFORGE	2954	0.83	0.8	0.83	0.82	0.91
JCP	1041	0.95	1	0.77	0.87	0.89
JEDIT	847	0.9	0.99	0.73	0.84	0.88
MOZILLA	496	0.92	1	0.73	0.85	0.87
PLONE	277	0.91	0.98	0.57	0.72	0.84
PSQL	2504	0.87	0.88	0.54	0.67	0.78
SVN	438	0.94	0.96	0.56	0.71	0.78
Average	1137	0.91	0.96	0.67	0.79	0.84

achieved using change classification with a reduced feature set. Table III reports, for each project, overall prediction accuracy, buggy precision, recall, F-measure, and ROC area under curve (AUC) for the Naïve Bayes classifier using F-measure optimized feature selection.

Observing these two tables, a striking result is that three projects with the Naïve Bayes classifier achieve a buggy precision of 1, indicating that all buggy predictions are correct (no buggy false positives). While the buggy recall figures (ranging from 0.54 to 0.83, with a average buggy recall of 0.69 for projects with a precision of 1) indicate that not all bugs are predicted, still, on average, more than half of all project bugs are successfully predicted.

Figures 1 and 2 summarize the relative performance of the two classifiers and compare against the prior work by Kim et al. Examining these figures, it is clear that feature selection significantly improves both accuracy and buggy F-measure of bug prediction using change classification. As precision can often be increased at the cost of recall and vice-versa, we compared classifiers using buggy F-measure. Good F-measure's indicate overall result quality.

Kim et al.'s results in both figures are taken from [1], where they were computed using the same corpus (with the exception of JCP, which was not in Kim et al.'s corpus and two projects which did not distinguish buggy and new features) using an SVM classifier trained on a set of features where no feature selection was performed (i.e., Kim et al.'s work used substantially more features for each project). Table II reveals the drastic reduction in the average number of features per project when using classifiers trained on a reduced feature set, as compared to Kim et al.'s prior work.

The additional benefits of the reduced feature set include better speeds of classification and scalability. We have noted

Fig. 1. Classifier Accuracy by Project

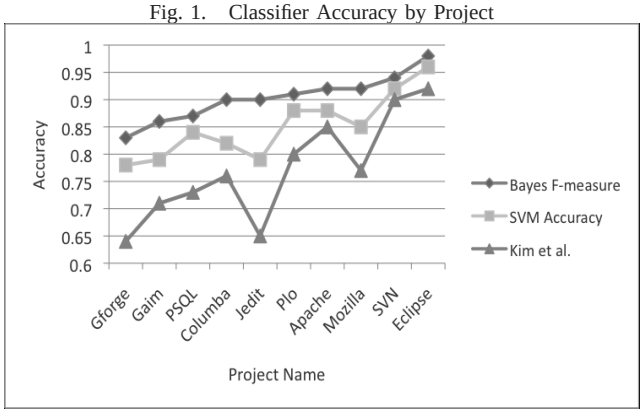
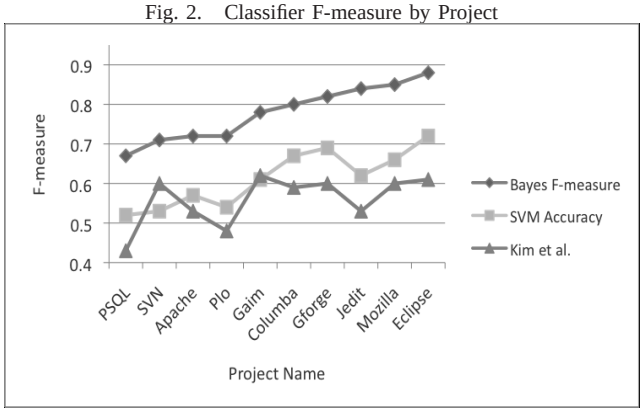


Fig. 2. Classifier F-measure by Project



a decrease in classification time from several seconds to a split second in many of the projects. This helps promote interactive use of the system within an Integrated Development Environment.

V. RELATED WORK

Given a software project containing a set of program units (files, classes, methods or functions, or changes depending on prediction technique and language), a bug prediction algorithm outputs one of the following.

Totally Ordered Program Units. A total ordering of program units from most to least bug prone [10] using an ordering metric such as predicted bug density for each file [11]. If desired, this can be used to create a partial ordering (see below).

Partially Ordered Program Units. A partial ordering of program units into bug prone categories (e.g. the top $N\%$ most bug-prone files in [11]–[13])

Prediction on a Given Software Unit. A prediction on whether a given software unit contains a bug. Prediction granularities range from an entire file or class [2], [14] to a single change (e.g., Change Classification [1]).

A. Totally Ordered Program Units

Khoshgoftaar and Allen have proposed a model to list modules according to software quality factors such as future

fault density using stepwise multiregression [10], [15], [16]. Ostrand et al. identified the top 20 percent of problematic files in a project [11] using future fault predictors and a linear regression model.

B. Partially Ordered Program Units

The previous section covered work which is based on total ordering of all program modules. This could be converted into a partially ordered program list, e.g. by presenting the top $N\%$ of modules as performed by Ostrand et al. above. Hassan and Holt use a caching algorithm to compute the set of fault-prone modules, called the top-10 list [13]. Kim et al. proposed the bug cache algorithm to predict future faults based on previous fault localities [12].

C. Prediction on a Given Software Unit

Using decision trees and neural networks that employ object-oriented metrics as features, Gyimothy et al. [14] predicted fault classes of the Mozilla project across several releases. Their buggy precision and recall are both about 70%, resulting in a buggy F-measure of 70%. Our buggy precision for the Mozilla project is around 100% (+30%) and recall is at 73% (+3%), resulting in a buggy F-measure of 85% (+15%). In addition they predict faults at the class level of granularity (typically by file), while our level of granularity is by code change, typically spanning only 20 lines of code. Aversano et al. [17] achieved 59% buggy precision and recall using KNN (K nearest neighbors) to locate faulty modules. Hata et al. [2] showed that a technique used for spam filtering of emails can be successfully used on software modules to classify software as buggy or clean. They achieved about 63.9% precision, 79.8% recall, and 71% buggy F-measure on the best data points of source code history for 2 eclipse plugins. We obtain buggy precision, recall, and F-measure figures of 98% (+34.1%), 79% (-0.8%) and 88% (+17%) respectively with our best performing technique on the eclipse project (Table III). Menzies et al [18] achieve good results on their best projects. However, on average the precision is low ranging from a minimum of 2%, a median of 20%, to a max of 70%. Both Menzies and Hata focus on the file level of granularity. Kim et al. showed that using support vector machines on software revision history information can provide an average bug prediction accuracy of 78%, a buggy F-measure of 60%, a precision and recall of 60% when tested on twelve open source projects [1]. Our corresponding results are an accuracy of 91% (+13%), a buggy F-measure of 79% (+19%), a precision of 96% (+36%), and a recall of 67% (+7%).

VI. Conclusion

This paper has explored the use of a feature selection algorithm to substantially decrease the number of features used by a machine learning classifier for bug prediction. An important pragmatic result is that feature selection can be performed in increments of 10% of all features, allowing it to proceed quickly. Between 4.1% and 12.52% of the total feature set yielded optimal classification results. The reduced feature set permits better and faster bug predictions.

The most important results in the paper are found in Table III, which present F-measure optimized results for the Naïve Bayesian classifier. It is astonishing that three projects have a buggy precision of 1, indicating no false positives in their bug predictions. The average buggy precision is .96, also very high. From the perspective of a developer receiving bug predictions on their work, these figures mean that if the classifier says a change has a bug, it is almost always right.

In the future, when software developers have advanced bug prediction technology integrated into their software development environment, the use of classifiers with feature selection will permit fast, precise, accurate bug predictions. With widespread use of integrated bug prediction, future software engineers can increase overall project quality in reduced time, by catching errors as they occur.

References

[1] S. Kim, E. W. Jr., and Y. Zhang, "Classifying Software Changes: Clean or Buggy?" *IEEE Trans. Software Eng.*, vol. 34, no. 2, pp. 181–196, 2008.

[2] H. Hata, O. Mizuno, and T. Kikuno, "An Extension of Fault-prone Filtering using Precise Training and a Dynamic Threshold," *Proc. MSR 2008*, 2008.

[3] J. Madhavan and E. Whitehead Jr, "Predicting Buggy Changes Inside an Integrated Development Environment," *Proc. 2007 Eclipse Technology eXchange*, 2007.

[4] A. Mockus and L. Votta, "Identifying Reasons for Software Changes using Historic Databases," *Proc. ICSM 2000*, p. 120, 2000.

[5] J. Sliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" *Proc. MSR 2005*, pp. 24–28, 2005.

[6] S. T. http://www.scitools.com/, "Maintenance, Understanding, Metrics and Documentation Tools for Ada, C, C++, Java, and Fortran," 2005.

[7] S. Scott and S. Matwin, "Feature Engineering for Text Classification," *Machine Learning-International Workshop*, pp. 379–388, 1999.

[8] E. Alpaydin, *Introduction To Machine Learning*. MIT Press, 2004.

[9] H. Liu and H. Motoda, *Feature Selection for Knowledge Discovery and Data Mining*. Springer, 1998.

[10] T. Khoshgoftaar and E. Allen, "Predicting the Order of Fault-Prone Modules in Legacy Software," *Proc. 1998 Int'l Symp. on Software Reliability Eng.*, pp. 344–353, 1998.

[11] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Trans. Software Eng.*, vol. 31, no. 4, pp. 340–355, 2005.

[12] S. Kim, T. Zimmermann, E. W. Jr., and A. Zeller, "Predicting Faults from Cached History," *Proc. ICSE 2007*, pp. 489–498, 2007.

[13] A. Hassan and R. Holt, "The Top Ten List: Dynamic Fault Prediction," *Proc. ICSM'05*, Jan 2005.

[14] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," *IEEE Trans. Software Eng.*, vol. 31, no. 10, pp. 897–910, 2005.

[15] T. Khoshgoftaar and E. Allen, "Ordering Fault-Prone Software Modules," *Software Quality J.*, vol. 11, no. 1, pp. 19–37, 2003.

[16] R. Kumar, S. Rai, and J. Trahan, "Neural-Network Techniques for Software-Quality Evaluation," *Reliability and Maintainability Symposium*, 1998.

[17] L. Aversano, L. Cerulo, and C. Del Grosso, "Learning from Bug-introducing Changes to Prevent Fault Prone Code," in *Proceedings of the Foundations of Software Engineering*. ACM New York, NY, USA, 2007, pp. 19–26.

[18] T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Trans. Software Eng.*, vol. 33, no. 1, pp. 2–13, 2007.