Bad Commit Smells

Those Who Do Not Learn From Their Commit History Will be Doomed to Revert it

Steven Johnson   Zach Welch

University of Wisconsin Madison

Abstract

On a given day, a large open−source Git Repository likely receives many pull requests, plenty of which are submitted by untrusted public members who have yet to contribute to the repository. With so much information to look up and code to test for managers of the repository, we have developed a method to extract metadata about commits and their messages, analyze this data using machine learning techniques, and determine which commits are suspect for introducing new bug(s). The accuracy of our classification method is verified using a number of open−source Git repositories, including jedit, jquery, scala, and Git itself. Our ensemble classifier correctly predicts bugginess of commits between 70 and 90% of the time, outperforming the baseline of random weighted guessing by 15 to 25%. This tool furthers the goal of better directing the testing efforts of repository owners by providing recommendations about suspect commits and also seeks to inform contributors in ways to improve their commit structure to make their commits less suspected.

December 16, 2013

Contents

List of Figures

List of Tables

Yeah Science!!

Jesse Pinkman

1 Introduction

Sometimes test suites simply aren't enough to ensure bug−free changes to code. Manual inspection by a human is, at times, necessary to ensure code correctness. Due to the large volume of commits submitted to open−source repositories each day, managers of these repositories often do not have the time to manually inspect all of these proposed changes to the code.

In this paper, we discuss our development of a model which attempts to predict the bugginess of commits submitted to a repository by using machine-learning techniques trained on attributes of the commits and their messages. This tool furthers the goal of better directing the testing efforts of repository owners by providing recommendations about which commits are most likely to be buggy, and it does so without any analysis of the code which is being committed.

Furthermore, a beneficial side−effect of our examination of commit attributes and commit message attributes is some prescriptive advice for future contributors to repositories. Examination of the information gain to our learner allowed us to see which attributes of commits were the most correlated with buggy commits, implying that perhaps contributors should alter the way they commit to avoid these attributes.

The next section frames this project with the related work of others. Section 3 discusses our approach in developing our model including data sources, feature selection, and algorithm development. Section 4 discusses the refinement of the model, presents our results, and discusses our methodology for the validation of our results. Section 5 discusses shortcomings and threats to validity, section 6 discusses future work, and section 7 concludes the paper.

2 Related Work

Using version control system history to aid the development process is not a new idea; the wealth of data that repositories store makes them very attractive artifacts to researchers who want to understand and improve the ways in which code is written. In the last ten years, a good deal of work has gone into trying to use repository data to make various aspects of the development process easier [3]. In the early 2000′s, Zimmerman et al. [13] and Ying [10] developed systems to mine version history to make suggestions on potential locations for changes based on recent alterations to versioned files. Their results were very interesting, reliably finding associations between sections of code and between code and documentation.

A common goal in many projects involving software repositories is attempting to predict the bugginess of future code using some set of metrics. A wide variety of metrics have been researched, including code churn [7], association of changed files [13], distance between team members [1], and commit size [2]. Recent research has also been done to analyze the accuracy of predicting one repository's future defects using a different repository's past history [11]. Work has also been done on analyzing characteristics of bugs and bug fixes [12].

There has also been some work in attempting to automatically infer which past commit introduced a bug. Originally described in [9], the SZZ algorithm uses keywords to find bug fixes, and then uses the version system's blame feature to find which commits last touched

the code this bug fix attempted to fix. Future improvements were also made by [5], which we discuss in detail later.

Some work has also been done by this group at attempting to use machine learning techniques to predict the bugginess of future commits [4]. Our work differs from theirs in several ways, namely that their work focuses more heavily on analysis of committed code, which we generally ignore in favor of the simple attributes and message structure of a commit. Superficially, their work was on Subversion projects while ours were on Git projects.

3   Approach

3.1   Mining Git Repositories

We mined eight opennon−trivialsource repositories hosted on Github.com for features to serve as our testing data sets, and these can be seen in Table 1. The repositories are quite varied; they range in size from around 2000-30000 commits and in age from roughly 3−13 years. The repositories also vary considerably in their goals − they include a programming language, a versioning control system, and a nuclear fuel cycle simulator. We extracted two sets of features about commits from each repository: commit attributes and message attributes. Commit attributes are information about a commit excluding actual analysis of the code or the commit message. Examples of commit attributes we extracted are the number of lines added, the number of lines removed, the contributor, the length of the comment, the day of the week of the commit, and the number of files altered. The commit message attributes we selected are the frequency of words, grammatical correctness, and readability and stylistic metrics of the message. We used the Flesch−Kincaid measure for readability [6] , diction, a unix command line tool, to quantify style (triteness, over−wordiness), and Queequeg to test the degree of grammatical correctness [8] for each commit message. Our analysis of the commit messages assumes they are written in English. A complete list of the attributes we examined can be found in Table 2.

3.2   Automatically Classifying Past Commits

To create a classification model from these features, we require a ground truth about whether or not each commit was bug inducing. We used a modified version of the SZZ algorithm from [9] to determine the set of commits for each repository that were bug inducing. This technique works by first identifying the set of commits in the repository which are bug fixes. Our modification of the algorithm accomplishes this with an analysis of the commit message looking either for words such as bug or fix or for strings of numbers and letters that appear to be bug−tracking numbers. After defining this set of bug fix commits, we process each commit individually, identifying sections of source code, called hunks, that the commit altered. For each hunk, we then flag as bug inducing the most−recent commit before the bug fix commit which altered that hunk of code. This analysis is conducted using an annotation graph search. Obviously, this heuristic is imperfect; it depends on bug fix commits being atomic and on bugs being discovered and fixed immediately after their introduction. However, as Kim et. al showed in [5], this technique tends to work well in practice, as their study

Repository	Startdate	# Commits	# Buggy Commits	% Buggy
one	June 2008	8179	2744	33.6
cyclus	March 2010	2162	1071	49.5
Git	April 2005	28788	10682	37.1
jedit	Sept 1998	6890	3190	46.3
jquery	March 2006	5608	3074	54.8
puppet	April 2005	12174	4812	39.5
scala	Feb 2003	20398	9584	47.0
scipy	Feb 2001	10035	3252	32.4

Table 1: Mined Repositories

Lines Added	Lines Added
Lines Removed	Lines Removed
Hour Commited (UTC)	
Minute Commited	
Day Commited	
Month Commited	
Year Commited	
Day of Week	Day of Week
Committing User	Committing User
Number of Files Commited	Number of Files Commited

Table 2: All Git Attributes (Left), Useful Git Attributes(Right)

using actual developers to confirm bugginess found only a 4-5% rate of false positives. This algorithm works well in practice and serves as a good estimation for which commits are bug inducing, so we adopted it.

3.3 Machine Learning Algorithms

With features extracted and a bugginess classification in hand for each commit for our selected repositories, we have the input prepared for constructing, training, and validating a machine learning algorithm. After some experimentation with individual classifiers, the method we selected to predict bugginess is an ensemble classifier, which aggregates the outputs of several machine learning classifiers (found in Table 3). Ensemble classifiers generally provide more accurate and consistent results than a single classifier because, assuming the individual classifiers are searching different parts of the hypothesis space, the likelihood of a majority of classifiers being wrong is less likely than a single classifier being incorrect. An added benefit of using an ensemble classifier is it easily allows the user to specify a confidence value for classifying commits as buggy; the more classifiers in agreement, the higher the confidence. Unless otherwise stated our results take the simple majority of the classifications returned.

weka.classifiers.trees.J48
weka.classifiers.bayes.NaiveBayes
weka.classifiers.functions.SMO
weka.classifiers.bayes.BayesNet
weka.classifiers.functions.MultilayerPerceptron
weka.classifiers.meta.Bagging
weka.classifiers.meta.ClassificationViaClustering
weka.classifiers.meta.RandomSubSpace
weka.classifiers.rules.ConjunctiveRule
weka.classifiers.rules.DecisionTable
weka.classifiers.trees.DecisionStump
weka.classifiers.functions.VotedPerceptron
weka.classifiers.rules.JRip

Table 3: List of Weka classifiers used

The tuning of this method involved an iterative process which included selecting the ideal set of features to train and test on, the sizes of the training datasets, and the machine learning algorithms and their parameters to serve as initial classifiers. We made sure to include at least one decision tree algorithm as part of our ensemble method due to the extra intuition which can be gained from an examination of the first few splits of its branches. Our prescriptive remarks to future repository committers were largely influenced by the output of these trees.

3.4 Pull Requests

In large open source projects, very rarely are commits added one by one. Instead, a developer checks out the main development branch, makes a series of changes, and then asks for their work to be merged back into the development branch (on Github, this is implemented as a pull request). This request contains a series of commits, many of which likely overlap in their changes and some which fix issues with earlier commits in the request (oops commits). Being able to classify an entire pull request as clean or buggy seems like it would be a much more applicable tool than one which simply classifies individual commits.

The simple, obvious solution would be to classify a pull request clean if all commits in it are clean, and buggy otherwise. This seems to be too drastic a measure, however, due to these oops commits. Any method for classifying requests should take into account that buggy commits in a pull request can potentially contain the associated bug fix in the same request. We propose a variation on the SZZ algorithm to handle this possibility. All commits are initially given a label using our ensemble classifier. For a buggy commit in the pull request, we take every hunk from the commit and look forward in time for a clean commit with that hunk. Finding a hunk in a clean commit would imply that whatever bug originally existed has been fixed. If all the buggy hunks have corresponding clean hunks later in the pull request, or if the number of unpaired buggy hunks is below some threshold, the pull request

is accepted as clean. Unfortunately, due to time constraints we were not able to test the effectiveness of this method.

4 Results

4.1 Feature Analysis

We went through several iterations of features to represent commits; we will briefly touch here on the features that did not work well before continuing to a more lengthy discussion of the features we actually used. Our first attempt at classifying commits attempted to cast this prediction problem as equivalent to catching email spam. A simple way to predict that an email is spam is to train on the individual words in spam emails. The more a word is associated with being seen only in spam email, the more likely an email containing that word is spam. We attempted to apply this to commits by training a Naive Bayes classifier on the words of a commit messages. This turned out to not be a very good idea, with results about as good as randomly guessing clean or buggy. Abandoning the buggy commit as email spam metaphor, we decided to look at attributes of the commit.

The features we chose to represent our commits are fairly simple ones; of the commit attribute data we use, only the day of the week required any computation. Most of these attributes are presented to the user by Git when they make a commit. We did some initial testing with a decision tree classifier, inspecting changes in results while testing different subsets of features. We also manually kept track of which features were pruned from the tree, which indicates they are less informative. We concluded time and date commit data is not very informative for the open source projects we chose to study. Year and Month are too large to contain much signal, and day of month and minute are too specific. Because hour is not stored in local time and the generally distributed nature of these projects, we did not find hour to be an informative feature. It is possible that these features could be more valuable in a highly centralized development structure, where there is extra knowledge of development schedules, but without any additional knowledge, we decided to ignore specific time−date data.

4.2 Testing Methodology

To test the effectiveness of our feature selection, we mined features from eight open source repositories. We attempted to choose projects with different life spans and applications to make our results more generalizable. The list of projects mined, their number of commits, and the percentage of those commits we found to be buggy are in Table 1.

For each repository we mined, we took the year of 2013, regardless of its size, to be our test set. The reasoning behind this is that in practice, users will want to classify new commits as they come in, and the most recent year of commits were the best fit of the data we had. Notice due to the way our heuristic works (start at a bug fix and work back) our test set is likely to contain more false negatives because the bug fixes have potentially not been written yet. However, in practice this will also be the case, so we feel this should not unreasonably skew our results.

Figure 1: Testing and Training Sets

Repo	Test Set	6Mo Train	1 Yr Train	2 Yr Train	All Train
one	2304	1190	2423	4367	5874
cyclus	546	389	1231	1524	1616
Git	2916	1369	2630	5129	25869
jedit	165	301	760	1401	6725
jquery	898	592	925	1934	4695
puppet	2195	980	2130	3854	9979
scala	1752	1347	2983	5080	18644
scipy	1873	546	1130	1786	8162

Table 4: Size of Training and Test Sets

Repo	Test Set	6Mo Train	1 Yr Train	2 Yr Train	All Train
one	23.1	26.4	30.2	35.4	37.7
cyclus	30.6	44.4	50.6	54.9	55.9
Git	13.9	22.1	22.5	23.4	39.7
jedit	6.7	13.9	24.7	23.1	47.3
jquery	35.7	43.2	46.6	53.9	58.6
puppet	18.7	20.0	26.1	29.1	44.1
scala	27.2	45.3	48.8	50.3	48.9
scipy	18.8	23.6	23.6	24.7	35.5

Table 5: Percent Bugginess of Training and Test Sets
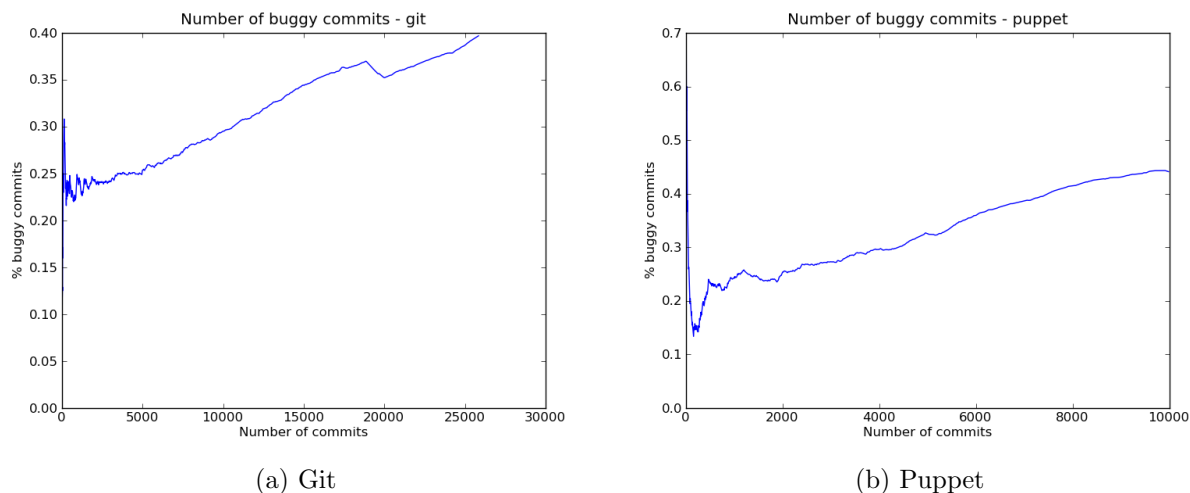
(a) Git

(b) Puppet

Figure 2: Percent Buggy Commits over Time

For sets of training data, we took commits from the final day of 2012 back six months, one year, two years, and to the start of the repository. The goal here is to see how our results change with more data to train on (and potentially overfit to). Tables 4 and 5 show the size and percentage of commits which are buggy, respectively. Notice that as time increases, the percentage of buggy commits increases (excepting scala all time). This seems to be an artifact of our heuristic; older bugs are more likely to have associated bug fixes, and so are more likely to be found by our method. Without having the actual distribution of buggy commits, we can make no assumptions about its trend over time.

To compare our features against some baseline, we created a random weighted coin classifier which calculated the percent buggy in a training set and used this percentage as the weight on a random buggy/clean coin for each test instance. We feel this represents a better floor against which to compare our results than simply using a fair coin. All discussion of random coin results are the average of 100 trials. To compare the effectiveness of attribute and message features, we also trained classifiers using only attribute features, only message features, and both sets of features.

4.3   Classifier Accuracy

Figure 3 contains our accuracy results of every mined repository using one year of training data. Results for random weighted coin vary but in general are about fifteen to twenty percentage points lower than the rest of the classifiers. Using attribute features only tends to work as well or slightly better than message features, with big drops only occurring in cyclus, jquery, and scala. Using both sets of features is comparable to only using attribute features. Variation between data sets is comparable to the percentage bugginess in each training set, which is to be expected.

As encouraging as these initial results are, accuracy may not be the optimal way to view this data; as Table 5 shows, the majority of our training and testing sets are less than 50% buggy. A classifier could conceivably classify all commits as clean, achieve greater than 50%
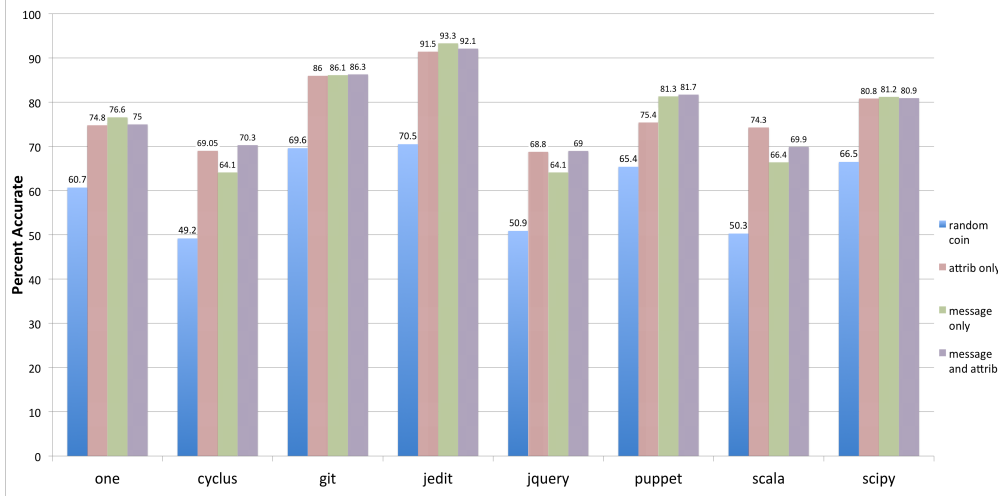
Figure 3: Comparison of Feature Results − 1 Year

accuracy, and be of no actual use in predicting buggy commits. In several instances, this is in fact what happens. The 93.3% accuracy seen by jedit is more than a little misleading; the jedit test set is only 6.7% buggy, and so the jedit classifiers simply guess clean for every instance. This is the case for the majority of message−only feature classifiers; only cyclus and jquery message−only classifiers ever classify any test commits as buggy. The results from cyclus and jquery are still promisingly higher than our baseline, which leads us to believe that our features do have some non−random correlation with clean and buggy commits. All of our attribute−trained and attribute/message−trained classifiers have non−trivial decision boundaries. To get a better idea of how well our classifiers actually are at correctly predicting commits as buggy, we examine precision/recall graphs for cyclus, scala, and scipy by varying the ensemble classifier threshold from 0.0% agreement (always classify buggy) to 100% agreement in 10% increments. Precision (True Positives/(True Positives+False Positives) and Recall (True Positives/(True Positives+False Negatives)) seem like much more natural fits to analyze how well our ensemble classifiers are at classifying buggy commits. We define buggy to be a positive and clean to be a negative. Intuitively, the farther up and to the right a point is, the better it performs at finding only true positives. We also provide some F-measure scores for the classifiers at various thresholds; the F-measure is the harmonic mean of precision and recall. Assuming precision and recall are equally valued, a higher F-measure is an indication of better performance.

As these figures show, using only comments can perform better than random, though not always without tuning of the threshold value. While at the default 50% threshold, Cyclus has a lower F-measure value than the random weighted coin; the threshold can can be tweaked to perform better. Scala performs better than average by default using message features than the random weighted coin. At the same time, using just comments performs noticeably worse in all cases than using attribute data. In every case we examined, attribute data outperforms comment data at all thresholds. Using attribute data performs about as well as training on both attribute and comment features, though attribute-only classifiers consistently contained the maximum F-measure. These results leads us to several conclusions. First, both sets of attribute and comment features are more predictive than random. Comparatively, attribute

9

(a) Cyclus

(b) Scala

(c) Scipy
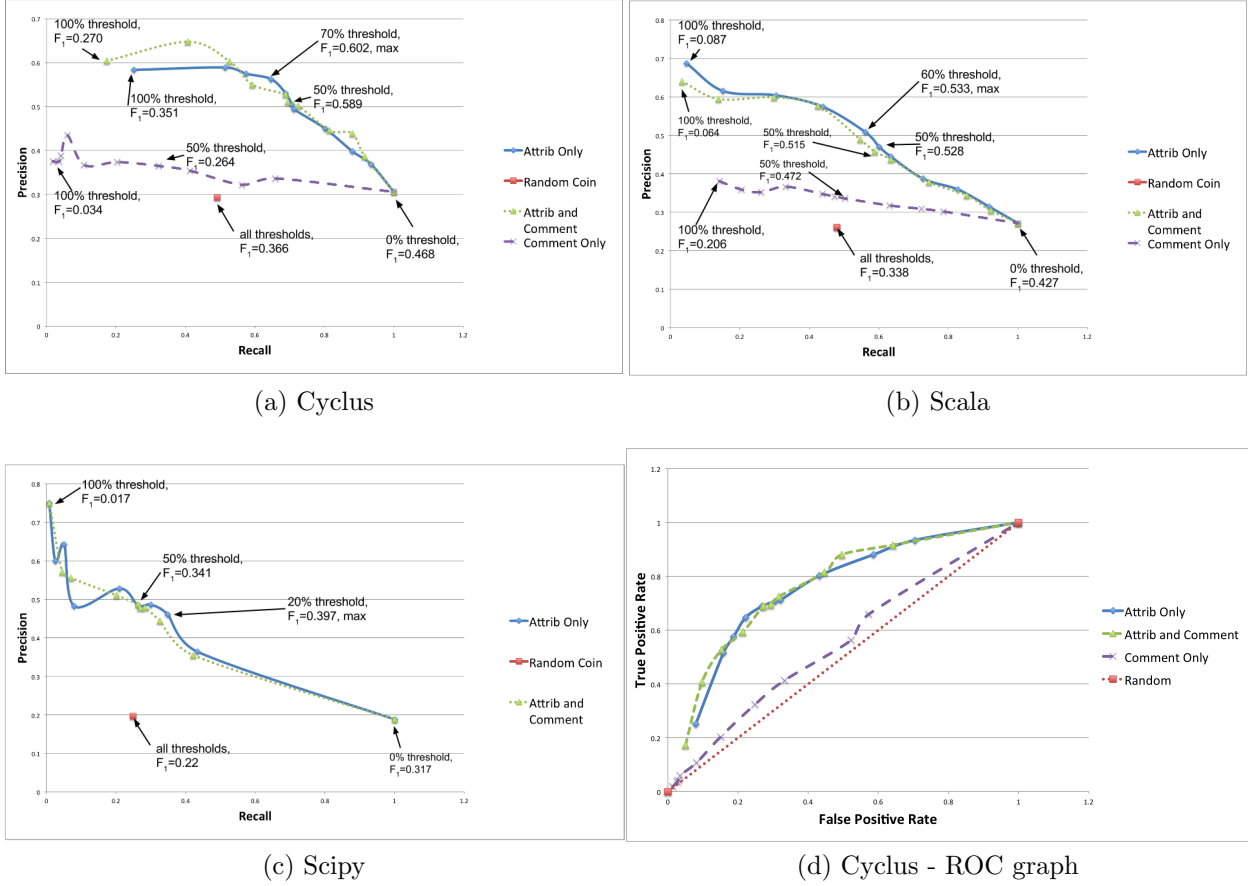
(d) Cyclus - ROC graph

Figure 4: Precision Recall Graphs

features are a much better approximation of a commit then comment features. Combining attribute and comment features provide little additional performance benefit over simply using attributes; with our current set of features, we do not gain substantive information from message features we did not already know from attribute features.

4.4 Training Set Variation

Another aspect of commits we wanted to look at is the size of the training set; is there a benefit to using older commits, or do they simply increase the noise in the data? If results are similar for a few months and the entire commit history, our classifier can be trained an order of magnitude faster with only a few months data. The accuracy of each classifier from six months to the beginning of the repository can be found in Figure 5a. Accuracy across repositories tended to trend slightly downward as the time frame from the training set increased. Not visible in Table 5 is the fact that for six months data sets, the likelihood of trivial decision boundaries increases. On the other end of the spectrum, however, using all the commit history seems to degrade results (possibly from overfitting). Some of these projects predate Git and almost certainly were originally in another version system. Looking more in depth, we examined the Precision Recall graph for Puppet in 5b. All time frames

10

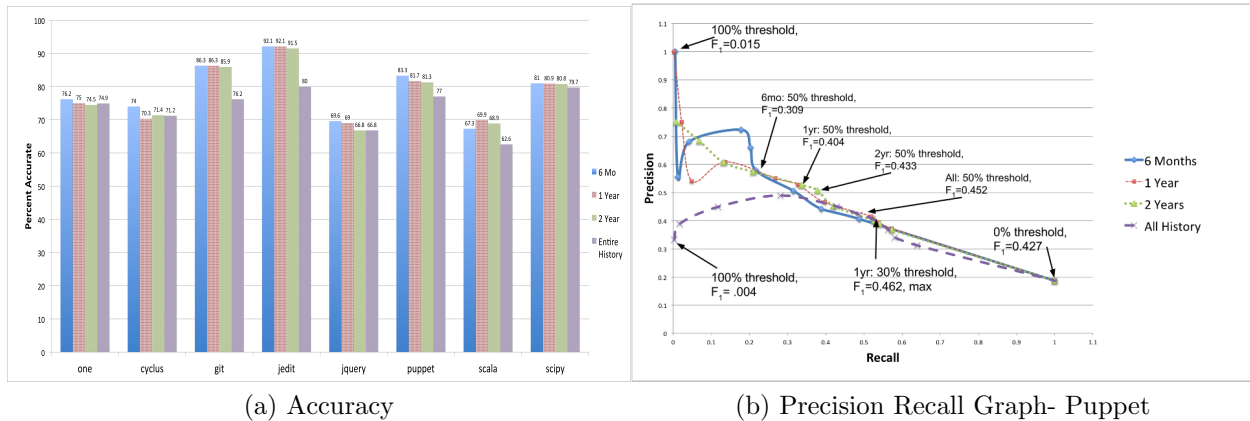(a) Accuracy　　　　　　　(b) Precision Recall Graph- Puppet

Figure 5: Training Data Size - Message and Attributes

perform similarly for smaller thresholds, with a very strong downward trend for the entire commit history approaching 100%. This would seem to imply that overfitting with all history only really becomes a problem when we force a large majority of the individual classifiers to agree. However, it appears that there is no real benefit from using more data − it just takes much longer to train.

Overall, it seems likely to us that recent commits are more representative of commits in the near future, which would explain the lack of change as we increase the size of the training set. If older commits contained more valuable information or significantly distorted the results, we would see the drop off in all results. From our work, we suspect six months to one year is the optimal range of training data, though this will obviously be different for each repository, depending on how often they make commits.

4.5　Cross Repository Training

A year of repository data before being able to use a tool seems like a high price of admission. Being able to substitute another repository′s history for your own with little to no performance cost would make a bug prediction tool more applicable. Furthermore, what does this say about commits if commits from two unrelated repositories are similar enough that one can be used to train the other? To answer these questions, we decided to use scipy and Git to train our classifier and test on the other repositories. Scipy and Git were chosen because they were large repositories with good performance and non−trivial decision boundaries. In order for training and testing data to be compatible, the user feature had to be removed from all data sets.

In general, results tended to be comparable to those using their respective data sets. This comparable performance indicates to us that using a classifier trained on another repository is a valid proxy for repositories that do not have their own history. It also seems that the distribution of message and commit attribute features we use do not vary much across repositories. More work would need to be done to give such a claim actual weight, but if our features have global, general boundaries, could we use this information in some kind of prescriptive fashion, a la keep your commits to X lines added if possible. Or would knowledge
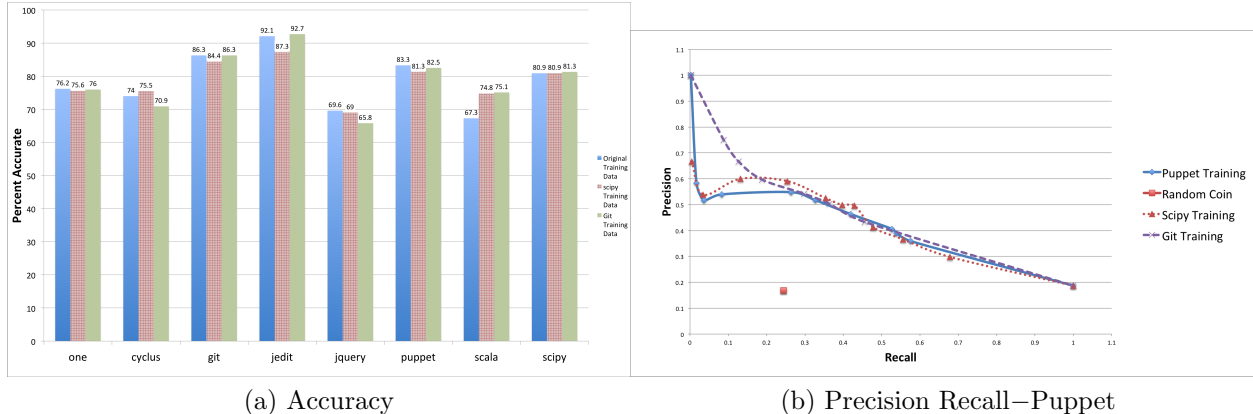
(a) Accuracy   (b) Precision Recall−Puppet

Figure 6: Training With Git and Scipy − 1 Months, Message and Attribute Data

and enforcement of this information simply shift the boundary?

5 Threats to Validity

There are several potential threats to the validity of our results. The heuristic used in classifying past commits represents a multifaceted threat. First, though Kim et. al [5] have presented some evidence that this method works well in practice, there is no evidence to support that these results generalize well. As we mined neither of the two projects they studied, it is possible that our approximation of the true distribution of clean and buggy commits is flawed. Another potential problem this heuristic introduces is that our ensemble learner is attempting to learn this approximation. If the approximation does not closely match the actual distribution, then the results we present are in no way an indication of how our tool would perform against actual commits.

Another potential threat is the repositories we chose. While we attempted to pick a representative sample of the open source projects in development, this excludes closed source projects. Also, we only mined english−speaking repositories, which is required by our heuristic classification and all of our message features. We made no attempt to take into account differences and eccentricities of the individual repositories we did mine. It is possible (even likely) that these eccentricities are to account for the variation between repositories.

6 Future Work

There are many potential avenues of future work that could stem from this work. One of the most important aspects of a commit that has purposefully been ignored in this work is the committed code itself. It is possible that a lightweight analysis of code changes could provide valuable indications of bugginess without making training our classifier intractable. Also, we believe looking at commit messages through the lense of more advanced natural language processing techniques could yield additional benefits.

Our results have shown commits are not especially repository specific, but we believe they could potentially be user−specific. Does it make more sense to have a user−specific classifier

that examines the user′s commit history across the repositories they have contributed to than to have a single classifier that predicts labels for potentially many users? Studying this would not require much additional new work to be done beyond mining new repositories and organizing training sets. If user−specific commits are in fact better predictors of bugginess, additional work would need to be done to ensure users do not cherry−pick their training data to give an overly positive but invalid bias.

We could have looked at varying the size of the testing set, or the distance between the end of the training set and the beginning of the test set to examine how often retraining would need to take place; a classifier that takes twelve hours to train itself every six months to maintain performance is a much different prospect than having to retrain it every few days. This information would be of great practical use to users and might also give us a better idea of which commits have the most impact on the decision boundaries.

Another potential way to approach this problem is through semi−supervised learning. Rather than attempting to classify all commits as clean or buggy, which requires automatic heuristics, the classifier could simply keep the distribution of commit features stored, some subset of which the user has manually classified as clean and buggy. The model could then be set up to request labels on past commits that it finds interesting or questionable.

7 Concluding Remarks

The bugginess of commit messages can be predicted with some level of accuracy. Commit messages also have some predictive information, though our current selection of message features performs worse than attribute features. Because buggy commits are often significantly less common than clean commits, our classifiers run the risk of potentially creating trivial classifiers that only predict clean. In practice, none of our attribute−feature−based classifiers experienced this, but several message−feature−based classifiers ran into this issue. Training using both attribute features and message features shows negligible benefit over training using only attribute features. After six months to a year, training on additional commits shows no performance boost and may in fact degrade results. We have shown that classifiers trained on one repository can be used to successfully predict the bugginess of commits from another repository with little−to−no degradation of performance.

References

[1] CATALDO, M., AND HERBSLEB, J. D. Factors leading to integration failures in global feature-oriented development: An empirical analysis. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, ACM, pp. 161–170.

[2] HINDLE, A., GERMAN, D. M., AND HOLT, R. What do large commits tell us?: A taxonomical study of large commits. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories* (New York, NY, USA, 2008), MSR '08, ACM, pp. 99–108.

[3] KAGDI, H., COLLARD, M. L., AND MALETIC, J. I. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol. 19*, 2 (Mar. 2007), 77–131.

[4] KIM, S., WHITEHEAD, JR., E. J., AND ZHANG, Y. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng. 34*, 2 (Mar. 2008), 181–196.

[5] KIM, S., ZIMMERMANN, T., PAN, K., AND WHITEHEAD, E. J. J. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 81–90.

[6] Microsoft office : Test your document's readability. `http://office.microsoft.com/en-us/word-help/test-your-document-s-readability-HP010354286.aspx`. Accessed: 2013-11-23.

[7] NAGAPPAN, N., AND BALL, T. Use of relative code churn measures to predict system defect density. In *Proc. Int'l Conf. Software Engineering (ICSE)* (2005), pp. 284–292.

[8] SHINYAMA, Y. Queequeg, a tiny english grammar checker. `http://queequeg.sourceforge.net/index-e.html`. Accessed: 2013-11-20.

[9] ŚLIWERSKI, J., ZIMMERMANN, T., AND ZELLER, A. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes 30*, 4 (May 2005), 1–5.

[10] YING, A. T. T., MURPHY, G. C., NG, R., AND CHU-CARROLL, M. C. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng. 30*, 9 (Sept. 2004), 574–586.

[11] ZIMMERMANN, T., NAGAPPAN, N., GALL, H., GIGER, E., AND MURPHY, B. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (New York, NY, USA, 2009), ESEC/FSE '09, ACM, pp. 91–100.

[12] ZIMMERMANN, T., NAGAPPAN, N., GUO, P. J., AND MURPHY, B. Characterizing and predicting which bugs get reopened. In *Proceedings of the 2012 International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE 2012, IEEE Press, pp. 1074–1083.

[13] ZIMMERMANN, T., WEIGERBER, P., DIEHL, S., AND ZELLER, A. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering* (May 2004), IEEE Computer Society, pp. 563–572.