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 roughly 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 suspect.

December 10, 2013

Contents

List of Figures

List of Tables

THIS SHOULD REALLY BE A QUOTE

ZACH

1 Introduction

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 2000s Zimmerman et al [1] and Ying [2] developed systems to mine version history to make suggestions on potential locations to make 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 [6], association of changed files, distance between team members [7], and commit size[8]. Recent research has also been done to analyze the accuracy of predicting one repositorys future defects using a different repositorys past history [12]. Work has also been done on analyzing characteristics of bugs and bug fixes[9] [10]

There has also been some work in attempting to automatically infer which past commit introduced a bug. Originally described in [CITE], the SZZ algorithm uses keywords to find bug fixes, and then uses the version systems blame feature to find which commits last touched the code this bug fix attempted to fix. Future improvements were also made by [CITE], 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. Our work differs from theirs in several ways, namely that their work focuses much more heavily on analysis of committed code, which we generally ignore in favor of the attributes and message of a commit.

3 Approach

3.1 Mining Git Repositories

We mined eight open-source repositories hosted on Github.com for features to serve as our testing data sets (Figure 1). The repositories are quite varied; they range in size from around 2000-30000 commits and in age from roughly 3-13 years. 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 Kincaid metric for readability [CITE], diction, a unix command line tool, to quantify style (triteness, over-wordiness) [CITE], and Queequeg to test the degree of grammatical correctness [CITE] for each commit message. Our analysis of the commit messages assumes they are written in English.

3.2 Automatically Classifying Past Commits

To create a classification model from these features, we required a ground truth about whether or not each commit was bug inducing. We used a modified version of a heuristic from [CITE] 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 heuristic accomplishes this with an analysis of the commit message looking 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 hunks of source code that the commit altered. For each hunk, we then flag as bug-inducing the most-recent commit before our bug fix commit which altered that hunk of code. This analysis is conducted using a annotation ASDFASDFASF (graph/tree) search [CITE]. Obviously, this heuristic is imperfectit depends on bug fix commits being atomic and on bugs being discovered and fixed immediately after their introduction. However, it works well in practice and serves as a good estimation for which commits are bug inducing, so we adopted it. (ASDFASDFSAF how do we know this????)

3.3 Features

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. The algorithm we selected to predict bugginess is an ensemble classifier, or in other words an aggregation of the outputs of several machine-learning algorithms (see Table 1). (ASDFASDFSDF list of our ml algs and perhaps their parameters??) The tuning of this method is an iterative process involving selecting the ideal set of features to train and test on, the sizes of both the training and testing datasets, and the machine-learning algorithms and their parameters to serve as initial classifiers. A list of features we examined, with those selected for the current version of our ensemble classifier highlighted, is found in Table 2.

3.4 Machine Learning Algorithms

3.5 Pull Requests

4 Results

4.1 Data

4.2 Analysis

4.3 Threats to Validity

5 Concluding Remarks

References