Automatic Classification of Large Changes into Maintenance Categories

Abram Hindle
University of Waterloo
Waterloo, Ontario
Canada
ahindle@cs.uwaterloo.ca

Daniel M. German
University of Victoria
Victoria, British Columbia
Canada
dmg@uvic.ca

Michael W. Godfrey
University of Waterloo
Waterloo, Ontario
Canada
migod@cs.uwaterloo.ca

Richard C. Holt
University of Waterloo
Waterloo, Ontario
Canada
holt@cs.uwaterloo.ca

Abstract

Large software systems undergo significant evolution during their lifespan, yet often individual changes are not well documented. In this work, we seek to automatically classify large changes into various categories of maintenance tasks — corrective, adaptive, perfective, feature addition, and non-functional improvement — using machine learning techniques. In a previous paper, we found that many commits could be classified easily and reliably based solely on the manual analysis of the commit metadata and commit messages (i.e., without reference to the source code). Our extension is the automation of classification by training Machine Learners on features extracted from the commit metadata, such as the word distribution of a commit message, commit author, and modules modified. We validated the results of the learners via 10-fold cross validation, which achieved accuracies consistently above 50%, indicating good to fair results. We found that the identity of the author of a commit provided much information about the maintenance class of a commit, almost as much as the words of the commit message. This implies that for most large commits, the Source Control System (SCS) commit messages plus the commit author identity is enough information to accurately and automatically categorize the nature of the maintenance task.

1 Introduction

Large commits are those in which a large number of files, say thirty or more, are modified and submitted to the Source Control System (SCS) at the same time. We demonstrated that large commits provide a valuable window into the software development practices of a software project [3]. For example, their analysis can highlight a variety of identifiable behaviours, including the way in which developers use the version control system (branching and merging), the incorporation of large amounts of code (such as libraries) into the code, and continual code reformatting.

While any large change might superficially seem to be significant due to its "size", however defined, it is important to recognize that not all large changes are created equal. For example, a commit that updates the stated year of the copyright for a system within the boilerplate comments might touch a very large number of files, but such a commit has no semantic impact on the system. Similarly, a change that performs a simple renaming of an oft-used identifier is less risky than a smaller but invasive refactoring of the program's design.

To aide further analysis, it is necessary to identify the type of change that a large commit corresponds to. For example, a developer who is interested in tracking how the code's design has evolved will probably want to ignore changes to whitespace, comments or licensing. This is particularly important for "basket-oriented" analysis of changes (for a survey of such methods see

[5]), where the inclusion of two entities, such as files, methods, or functions, in the same commit is expected to be significant. The relationship between two files that were changed during a change in whitespace, perhaps while reformatting of the source code, is probably negligible when compared to the relationship of two files changed during refactoring. Classifying large changes is useful to select or to ignore certain large changes during the analysis and exploration of such changes.

A major challenge, however, is how to automatically classify such large commits. The simplest solution would be for developers to manually tag commits with metadata that indicates the type of commit performed. While this is feasible in an organization that dictates software development practices, it does not provide a solution to the large corpus of commits already performed.

Another solution is to manually classify previous large commits. For a given project this may be feasible. Large commits correspond usually to the top 1% of commits with the most files changed. However, a manual approach may not scale well for some systems; consequently, we decided to investigate automatic classification of large changes.

When a programmer commits a change, they may disclose the intention of a change in the commit message. Often these commit messages explain what the programmer did and what the intended purpose of the change was. If it is a large change one might expect a programmer to document the change well.

We show in this paper that large source control system (SCS) commit messages often contain enough information to determine the type of change occurring in the commit. This observation seems invariant across different projects that we studied, although to a different extent per each project.

Classifying commits using only their metadata is attractive because it does not require retrieving and then analyzing the source code of the commit. Retrieving only the meta-data of the commit is significantly less expensive, and allows for efficient browsing and analysis of the commits and their constituent revisions. These techniques would be useful to anyone who needs to quickly categorize or filter out irrelevant revisions.

2 Previous Work

This work extends our work on Large Commits classification [3] and the classification work of Robles et al. [2]. We rely on the data produced by our Large Commits study to execute this study.

These works derive from Swanson's Maintenance Classification [7], that provided a taxonomy for maintenance changes (see Table 1), the study of small changes by Purushothan et al. [6], and later work by Alali et al. [1]. Purushothan et al. extended the Swanson Maintenance Classification to include code inspection, then they classified many small changes by hand and summarized the distribution of classified revisions. Alali et al. further studied the properties of commits of all sizes but did not classify the commits. Our previous work classified large commits by their maintenance type; we discovered that many large commits are integration and feature addition commits, which were often ignored as noise in other studies. We made multiple categorization schemes similar to Swanson (see Table 2 for the Large Changes categorization). Robles et al. had a similar idea and extended the Swanson categorization hierarchically.

Categories of Change	**Issues addressed.**
Corrective	Processing failure Performance failure Implementation failure
Adaptive	Change in data environment Change in processing environment
Perfective	Processing inefficiency Performance enhancement Maintainability
Feature Addition	New requirements
Non functional	Legal Source Control System management Code clean-up

Table 1. Our categorization of maintenance tasks; it extends Swanson's categorization [7] to include feature addition and non-functional changes as separate categories. We refer to this as the Extended Swanson Categories of Changes.

3 Methodology

Our methodology can be summarized as follows. We selected a set of large, long-lived open source projects; for each of them we manually classified 1% of their largest commits, the commits with the most files changed. We used these manually classified commits to determine if machine learning was useful to automatically classify them. The details are described in this section.

Categories of Large Commits	Description
Feature Addition	New requirements
Maintenance	Maintenance activities.
Module Management	Changes related to the way the files are named and organized into modules.
Legal	Any change related to the license or authorship of the system.
Non-functional source-code changes	Changes to the source code that did not affect the functionality of the software, such as reformatting the code, removal of white-space, token renaming, classic refactoring, code cleanup (such as removing or adding block delimiters without affecting the functionality)
SCS Management	Changes required as a result of the manner the Source Control System is used by the software project, such as branching, importing, tagging, etc.
Meta-Program	Changes performed to files required by the software, that are not source code, such as data files, documentation, and Makefiles.

Table 2. Categories of Large Commits: they better reflect the types of large commits we observed than those by Swanson.

3.1 Projects

The projects that we selected for this study were widely used, mature, large open source projects that spanned different application domains (and were representative of each of them). These projects are summarized in Table 3. The projects are also implemented in a variety of programming languages: C, C++, Java and PHP. Some projects were company sponsored, such as MySQL and Evolution.

3.2 Creating the training set

For each project we extracted their commit history from their corresponding version control repositories (CVS or Bitkeeper). We ranked each commit by number of files changed in a commit and selected the top 1% of these commits. We manually classified about 2000 commits into the Extended Swanson Classification (described in Table 1), Large Changes classification (see Table 2), and detailed large commits classification. This process is described in detail in [3].

We took this annotated dataset and wrote transformers to generate the datasets we wanted to use for automatic classification. We produced datasets such as the Word Distribution dataset and the Author, File Types and Modules dataset.

3.3 Features used for classification

We used the following features for the datasets used to train the classifiers. Each feature was extracted for each project.

Word Distribution By word distribution we mean the frequency of words in the commit message. We wanted to know if Bayesian type learners were useful for automatic classification. Bayesian learners are used in email spam filters to discriminate between SPAM (undesirable) and HAM (desirable) type messages. These learners determine the probability of a word distribution of a message (the word counts) occurring in both the Ham and SPAM datasets. Per each large commit we counted the number of occurrences of each word and the number of files that changed in that commit. We kept only tokens consisting of letters and numbers, and no other stop words were removed.

Author The identity of the commit's author. We suspected that some authors were more likely to create certain types of large commits than others.

Module and File Types Is the kind of commit affected by the module, that is the directory where the commit primarily occurred? These features were counts of files changed per directory, thus given a system each directory would become a feature of file counts per that directory. Along with a count of files changed per directory, we also count the files by their kind of usage: source code, testing, build/configuration management, documentation and other (STBDO [4]). These counts are represented by five respective features that are the counts of changed files for each type.

For each project, we created datasets from each of these features and their combinations (for example, one dataset was created for the word distribution and authors; another for all the features). We also created a dataset with the words distributions in the commit

Software Project	Description
Boost	A comprehensive C++ library.
Egroupware	A PHP office productivity CMS project that integrates various external PHP applications into one coherent unit.
Enlightenment	A Window Manager and a desktop environment for X11.
Evolution	An email client similar to Microsoft Outlook.
Firebird	Relational DBMS gifted to the OSS community by Borland.
MySQL (v5.0)	A popular relational DBMS (MySQL uses a different version control repository for each of its versions).
PostgreSQL	Another popular relational DBMS
Samba	Provides Windows network file-system and printer support for Unix.
Spring Framework	A Java-based enterprise framework much like enterprise Java beans.

Table 3. Software projects used in this study.

messages for the union of all commits for all projects. Authors and modules were too specific to each project to be useful when considering all the commits. This would permit us to determine if the automatic classification was applicable to all the projects, or if it worked better for some than for others.

3.4 Machine Learning Algorithms

Each of these datasets was fed into multiple machine learning algorithms. We wanted to know which algorithms performed better across the different projects, or if different algorithms were better for different projects). We used the various machine learners from the Weka Machine Learning framework [8]. We use 10-fold cross validation to train a model on 90% of the data and then test the other 10% against this created model. Thus for each learner it created 10 different models and executed 10 different test runs. In total, we used seven Machine Learners:

- J48 is a popular tree-based machine learner (C4.5). It makes a probabilistic decision tree, that is used to classify entities.
- NaiveBayes is a Bayesian learner similar to those used in email spam classification.
- SMO is a support vector machine. Support Vector Machines increase the dimensionality of data until the data points are differentiable in some dimension.
- KStar is a nearest neighbor algorithm that uses a distance metric like the Mahalanobis distance.
- IBk is a single-nearest-neighbor algorithm, it classifies entities taking the class of the closest associated vectors in the training set via distance metrics.
- JRip is an inference and rules-based learner (RIPPER) that tries to come up with propositional rules which can be used to classify elements.
- ZeroR is a learner used to test the results of the other learners. ZeroR chooses the most common category all the time. ZeroR learners are used to compare the results of the other learners to determine if a learners output is useful or not, especially in the presence of one large dominating category.

4 Results

The datasets were run against each of the machine learners. We used five metrics to evaluate each learner: *% Correct* is what percentage of commits were properly classified, which is both the recall and the accuracy; *% Correct ZeroR* was the accuracy of the ZeroR classifier against the same dataset — it is expected that a learner that is classifying data well should always do better than ZeroR; Δ ZeroR is the difference between the *%Correct* and *%Correct Zero* accuracies; *F-1* is the F-Measure, a value between 0 and 1 produced by a combination of precision (instances were not misclassified) and recall (total correctly classified instances); and *ROC*, which is the area under the Receiver Operating Characteristic (ROC) curve–this value is similar to the F-Measure and it is based on the plotting of true positives versus false positives, yet the end value closely mimics letter grades in the sense of the quality of classification, e.g., 0.7, a *B* grade, is considered fair to good.

Table 4 shows the results of the best learner using the dataset *Words distributions* per project. As it can be observed, no single classifier is best; nonetheless, usually the results were better than the ZeroR classifier which indicates that some useful classification was being done by the learner. Table 5 shows that the results are less accurate for the *Authors and Modules* dataset, but still better than ZeroR.

The *Word Distribution* dataset appears to be better at producing successful classifiers than the second *Authors and modules* dataset.

4.1 Learning from Decision Trees and Rules

The learners provided valuable output about their models and their classifications. Some of them helped by highlighting the most useful information they use to do the classification. In particular, the output of tree- and rules-based learners provides some interesting insights.

The J48 tree learner produced decision trees based on the training data. The trees created for the word distribution datasets are composed of decisions that depend on the existence of specific words, such as *cleanup* or *initial*. Specific authors and directories were also commonly used by decision in these trees.

The decision trees output by the J48 learner contain one word in each of its nodes; we tabulated such words and used their frequency to create text clouds. Figure 1 depicts the text cloud of words used in the union of all projects, while Figure 2 corresponds to the text clouds for each project. Words related to implementation changes were the most frequent, such as “add”, “added”, “new”. Other frequent words relate to *refactoring*, *bug fixing*, *documentation*, *cleanup* and *configuration management* changes.

We believe this suggests that simple word match classifiers can automate the previously manual task of classification.

The JRip classifier is a rules-based classifier that infers a small set of rules used to classify instances. With the word-based dataset, JRip produced rulesets which contained both expected rules and sometimes unexpected rules. The unexpected rules were often the most interesting.

For the large-changes classification, if the commit message contained words such as “license” or “copyright”, the commit message would be classified as a *Legal* change. If the commit message contained words such as (or related to) “fixes”, “docs” and “merging” it was classified as *Maintenance*. If the commit message contained words such as “initial” or “head” it was classified as *Feature Addition*. “Translations”, “cleanup”, “cleaning”, “polishing” and “whitespace” were words associated with *Non-functional* changes.

One interesting result was that for PostgreSQL, the classifier properly classified commit messages that contained the word “pgindent” (the name of its code indenter) as *Non-Functional*.

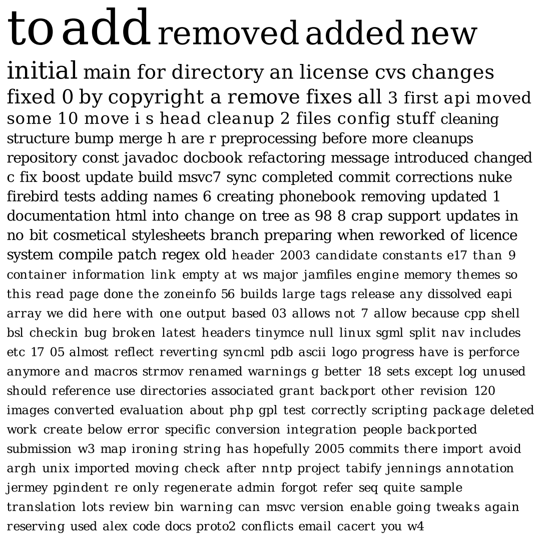

Figure 1. Text Cloud of tokens used by the J48 learner to classify commits. The size of the text indicates how many instances of the word occurred in trees produced for the union of all the projects’ commits.

For the dataset of *Authors, File Types and Directories*, we found that author features were most frequently part of rules. For instance in Firebird “robocop” was associated with *Non-functional* changes and “skywalker” was associated with *Module-Management*, in Boost, “hkaiser” was associated with *SCS-Management*. The counts of file types such as “Source”, “Test”, “Build”, “Documentation” and “Other” we used in some decision rules. Often changes with low counts of “Other” files were believed to be *Non-functional* changes, but those with many “Other” files were considered *Module-Management* changes.

For the Extended Swanson classification on the *Word Distribution* based dataset, we witnessed that *Non Functional* changes were associated with words such as “header”, “license”, “update”, “copyright”. *Corrective* changes had words such as bug, fixes and merge, while *Adaptive* had words such as Java-doc, zoneinfo, translation, build and docs. *Perfective* was a large category and it contained many refactoring and cleanup related words such as: “package”, “reorganized”, “nuke”, “major”, “refactoring”, “cleanup”, “removed”, “moved”, “pgindent” and “directory”.

Feature Addition was a large category that was often used as the default choice by the classifiers when no other rules matched. For the other projects, it was associated with the words: “initial”, “spirit”, “version”, “checkin”, “merged”, “create”, “revision” and “types”.

For the *Author and Module* dataset we noticed a few authors associated with *Perfective* changes. Only two authors were associated with *Non-functional* and *Feature Addition* changes, both were part of the Boost project, and none with *Adaptive* changes. This might suggest that some authors serve a support role of cleanup, or simply spend many of their commits cleaning up the source code.

4.2 Authors

On the *Author and Modules* dataset we applied attribute selection to try to find the attributes that mattered the most for classifying and discriminating between different types of commits. The Author attribute ranked the best for almost all of the projects. Combining the textual features and the author features might result in better learners.

Using the *Word Distribution and Authors* dataset, and *Authors and Modules* the results often improved slightly for most categorizations (1 to 3% improvement in accuracy). This suggests that authors and word distributions might be correlated. Some authors might be using similar language in their commit messages or the have specific roles that make them more likely to perform certain types of commits.

4.3 Accuracy

Table 4 indicates that for Swanson classifiction the projects of PostgreSQL, Evolution, Egroupware, Enlightenment, and Spring Framework had good accuracies (% Correct) above 60%, and usually good to fair ROC values of above 0.7. Evolution, Spring Framework and Firebird had the most significant improvements in accuracy compared with the ZeroR classifier applied to the same projects. These higher accuracies held for the large commits and detailed commits classifications as well, but were not as prominent. The lower accuracy in comparison to the Swanson ones, was probably because the other two classifications were more fine grained which can add more margin for error.

The classifiers did not work well for MySQL. We suspect this is because many of the large MySQL changes were merges and integrations, their comments were often about version control related issues, that might have been automatically generated.

The only significant failure in classifying was for Samba and our Large Classification categorization on the word distribution dataset. ZeroR beat JRip and NaiveBayes by 4% accuracy. As expected, the recall for ZeroR was quite poor (as seen in $F-1$ measure in Table 4). This result was not repeated in the Authors and Modules dataset, shown in Table 5, where J48 edged out ZeroR by 2%.

The *Author and Module* dataset usually did worse than the *Word Distribution* dataset, which suggests that the author of a commit and the modified modules provides almost the same, or as much information as the word distribution. Enlightenment was a notable case where the *Author Module* dataset worked out better than the *Word Distribution* dataset. Table 5 summarizes the best results from the *Authors and Modules* dataset.

When we combined the datasets of *Word Distributions*, *Authors* and *modules*, we found that usually, for most categorizations except the Swanson classification that the Authors/Modules/Word Distribution dataset had the best accuracy, but often its ROC and F-Measure scores were lower implying that there was some loss of precision. The information gained from adding the module features to the word distribution is low, as shown by the *Word Distribution and Author* dataset versus the *Word Distribution, Module and Author* dataset.

Finally, we averaged the results of all the datasets by weighting the number of commits for each project in such a way that each project contributed equally to the average. This result is shown in Table 6. We were surprised that usually most datasets provide very similar results. This suggests, while universal classifiers are useful, they are not as effective as those that are trained with a dataset from the project in question.

4.4 Discussion

What makes a good training set? Usually larger training sets produce better the results. One observation we made was for those projects where we used fewer annotations per change (only one annotation per change), and we summarized the change succinctly, those projects usually had greater classification accuracy. The better accuracy is probably because the classes would overlap less.

What are the properties of a project with high classification accuracy? Projects that use consistent terminology for describing certain kinds of SCS operations, and for describing changes made to a system will have a higher accuracy. Our study has shown that only a few unique words indicate the likely class. Short commit messages and inconsistent terminology would make classification more difficult.

The terminology used in the commit messages seems to provide as much information as the identity of the commit author provides. Since the author creates the commit message, perhaps the machine learners are learning the commit message style or the language of an author's role, rather than a project-wide lexicon or idiom.

Table 4. Best Learner Per Project for Word Distributions

Category	Project	Learner	% Correct	% Corr. ZeroR	Δ ZeroR	F-1	ROC
Ext. Swanson	Boost	J48	51.84	37.12	14.72	0.51	0.71
	EGroupware	JRip	66.18	64.18	2.00	0.54	0.53
	Enlightenment	SMO	60.00	53.81	6.19	0.56	0.71
	Evolution	SMO	67.00	44.00	23.00	0.64	0.73
	Firebird	J48	50.06	34.49	15.57	0.49	0.73
	MySQL 5.0	JRip	35.04	29.91	5.13	0.26	0.55
	PostgreSQL	NaiveBayes	70.10	55.67	14.43	0.69	0.82
	Samba	NaiveBayes	44.26	43.03	1.23	0.37	0.66
	Spring Framework	SMO	62.76	43.54	19.22	0.61	0.76
	Union of all projects	J48	51.13	38.85	12.28	0.50	0.74
Large Commits	Boost	JRip	43.13	33.07	10.06	0.38	0.65
	EGroupware	JRip	43.82	40.18	3.64	0.31	0.52
	Enlightenment	J48	44.05	39.76	4.29	0.36	0.56
	Evolution	IBk	54.00	39.00	15.00	0.45	0.65
	Firebird	J48	36.40	25.94	10.45	0.33	0.66
	MySQL 5.0	JRip	31.20	31.20	0.00	0.15	0.47
	PostgreSQL	SMO	68.04	52.58	15.46	0.64	0.76
	Samba	ZeroR	42.74	42.74	0.00	0.26	0.48
	Spring Framework	JRip	40.72	38.02	2.69	0.29	0.55
	Union of all projects	J48	38.97	24.42	14.54	0.38	0.69
Detailed	Boost	J48	27.82	17.44	10.38	0.25	0.67
	EGroupware	JRip	24.91	19.61	5.30	0.14	0.57
	Enlightenment	JRip	21.41	18.00	3.42	0.11	0.51
	Evolution	SMO	51.00	24.00	27.00	0.46	0.63
	Firebird	NaiveBayes	18.95	11.35	7.60	0.16	0.68
	MySQL 5.0	JRip	17.81	17.81	0.00	0.05	0.44
	PostgreSQL	SMO	61.62	48.48	13.13	0.54	0.60
	Samba	NaiveBayes	34.43	31.53	2.90	0.31	0.68
	Spring Framework	JRip	15.22	14.13	1.09	0.06	0.53
	Union of all projects	J48	23.42	10.71	12.71	0.22	0.71

Table 5. Best Learner Per Project for Authors and Modules

Category	Project	Learner	% Correct	% Corr. ZeroR	Δ ZeroR	F-1	ROC
Ext. Swanson	Boost	J48	41.85	37.22	4.63	0.40	0.64
	EGroupware	JRip	64.79	64.07	0.73	0.52	0.51
	Enlightenment	J48	66.51	53.68	12.83	0.63	0.74
	Evolution	SMO	67.01	42.27	24.74	0.66	0.78
	Firebird	J48	48.07	34.45	13.62	0.47	0.72
	MySQL 5.0	JRip	34.48	30.17	4.31	0.23	0.52
	PostgreSQL	J48	62.50	55.00	7.50	0.60	0.66
	Samba	JRip	45.81	42.94	2.86	0.42	0.64
	Spring Framework	SMO	55.09	43.41	11.68	0.53	0.65
Large Commits	Boost	JRip	36.84	33.17	3.67	0.24	0.55
	EGroupware	J48	40.29	40.11	0.18	0.29	0.52
	Enlightenment	J48	52.49	39.67	12.83	0.43	0.68
	Evolution	SMO	57.73	37.11	20.62	0.56	0.75
	Firebird	J48	32.45	25.91	6.54	0.31	0.66
	MySQL 5.0	JRip	30.60	30.60	0.00	0.14	0.47
	PostgreSQL	NaiveBayes	62.50	51.25	11.25	0.61	0.74
	Samba	J48	44.69	42.65	2.04	0.36	0.61
	Spring Framework	JRip	39.40	37.91	1.49	0.24	0.52
Detailed	Boost	JRip	19.37	17.57	1.80	0.08	0.52
	EGroupware	JRip	20.63	19.58	1.06	0.08	0.51
	Enlightenment	JRip	27.27	17.95	9.32	0.16	0.56
	Evolution	J48	47.42	21.65	25.77	0.46	0.73
	Firebird	J48	13.79	11.33	2.45	0.12	0.67
	MySQL 5.0	JRip	17.96	17.96	0.00	0.06	0.44
	PostgreSQL	SMO	60.98	47.56	13.41	0.56	0.67
	Samba	SMO	31.47	31.47	0.00	0.30	0.72
	Spring Framework	JRip	14.09	14.09	0.00	0.04	0.48

Table 6. Best Average Learner for each dataset. As expected, the more information, the better. However, the Frequency of Words and the Authors appear to be the most significant contributors.

Dataset	Category	Learner	% Correct	% Corr. ZeroR	Δ ZeroR	F-1	ROC
Frequency of Words (Words)	Ext. Swanson	SMO	52.07	44.46	7.61	0.50	0.68
	Large Commits	JRip	41.37	36.69	4.68	0.32	0.57
	Detailed	JRip	25.71	21.31	4.40	0.17	0.56
Authors and Modules	Ext. Swanson	J48	50.84	44.80	6.04	0.47	0.63
	Large Commits	JRip	41.24	37.60	3.64	0.31	0.57
	Detailed	JRip	25.01	22.13	2.88	0.15	0.54
Authors and Words	Ext. Swanson	SMO	53.27	44.80	8.47	0.51	0.70
	Large Commits	JRip	42.61	37.60	5.01	0.33	0.58
	Detailed	JRip	27.06	22.13	4.93	0.18	0.56
Authors, Words and Modules	Ext. Swanson	JRip	53.51	44.80	8.70	0.47	0.62
	Large Commits	JRip	43.20	37.60	5.60	0.34	0.60
	Detailed	JRip	27.38	22.13	5.25	0.19	0.57
Words and Modules	Ext. Swanson	J48	52.59	44.80	7.79	0.50	0.66
	Large Commit	J48	43.29	37.60	5.69	0.39	0.63
	Detailed	JRip	27.71	22.13	5.58	0.18	0.57
Author	Ext. Swanson	SMO	51.27	44.80	6.47	0.45	0.63
	Large Commit	J48	41.82	37.60	4.22	0.34	0.60
	Detailed	SMO	27.05	22.13	4.92	0.21	0.61
Modules	Ext. Swanson	J48	51.55	44.80	6.75	0.48	0.63
	Large Commit	JRip	41.57	37.60	3.97	0.30	0.56
	Detailed	JRip	24.78	22.13	2.65	0.15	0.53

Projects whose developers have consistent roles will probably fare better as well. A project with developers dedicated to code cleanup, or repository maintenance will prove easier to classify than authors of a team who shares responsibilities.

The lesson learned from this is the more that developers annotate their changes consistently the more likely that we can categorize the changes. This also implies that maintaining a consistent lexicon for a project will result in better automation of classification tasks.

5 Validity

For ground truth of categorizations we relied solely on annotations that we performed on the data. We also modified and created categorizations which threatens the objectivity of the ground truth. We might be discovering an automation of how we annotated the changes as we simply automated our manual classification in our first study.

The use of multiple annotations per change probably reduced the accuracy of the classifiers. We think that if the annotations were single purpose, we would get better results. Alternatively we could have asked the classifier to produce probabilities per each category.

A potential pitfall of this technique is that if it is continuously used throughout development, the number of features will increase as the word distribution increases, which happens as new tokens are introduced.

6 Future Work

Future work includes integrating this work into developer tools used to view and browse commits, where classifying commits would be relevant to developers. A direct extension to our work would be to test if using the source code tokens provide any benefit for classifying commits. We would also want to generalize further and investigate commits of all sizes.

7 Conclusions

Our results indicate that commit messages provide enough information to reliably classify large commits into maintenance categories. We applied several machine learners and each learner indicated that there was some consistent terminology internal and external to projects that could be used to classify commits by their maintenance task. We have showed that the arduous task of classifying commits by their maintenance activity, which we carried out in our previous work [3], can be automated.

We have shown that the author's identity may be significant for predicting the purpose of a change, which suggests that some authors may take on a role or take responsibility for a certain aspect of maintenance in a project.

We have been able to automate the classification of commits, using the commit messages, that we previously manually applied. We have shown that commit messages and the identity of authors provide learners with enough information to allow most large commits to be classified automatically.

Acknowledgements: We thank Gregorio Robles for the initial direction of this work.

References

[1] A. Alali, H. Kagdi, and J. I. Maletic. What's a typical commit? a characterization of open source software repositories. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 182–191, Washington, DC, USA, 2008. IEEE Computer Society.

[2] J. J. Amor, G. Robles, and J. M. Gonzalez-Barahona. Discriminating development activities in versioning systems: A case study. In *Proceedings PROMISE 2006: 2nd. International Workshop on Predictor Models in Software Engineering*, 2006.

[3] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108, New York, NY, USA, 2008. ACM.

[4] A. Hindle, M. W. Godfrey, and R. C. Holt. Release Pattern Discovery via Partitioning: Methodology and Case Study. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 19, Washington, DC, USA, 2007. IEEE Computer Society.

[5] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, 2007.

[6] R. Purushothaman. Toward understanding the rhetoric of small source code changes. *IEEE Trans. Softw. Eng.*, 31(6):511–526, 2005. Member-Dewayne E. Perry.

[7] E. B. Swanson. The Dimensions of Maintenance. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[8] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, San Francisco, USA, 2nd edition, 2005.

move config license merge removed fixes proto2 refactoring cvs 0 checkin updates 1 2 copyright almost cosmetical merged to perforce specific fixed licence and cpp 4 9 3 link tabs borland creating revision by preprocessing head converted proposal copyrights directory review ascii v1 part progress sync documentation into 295 regex commit change line docs submission gif initial tests candidate phoenix2 message bump hpp unix contest friendly 6 empty moving problem apply library pre add dunno headers support graph jamfiles class building separate main default for removing links cleanup seq doc updated minmax bbc2 accessor tr2 miniboost boost a remove update reverting style tweaks reworked linefeeds from good is all fusion attributes again reserving dots name renamed also cr on

(a) Boost

head added fixed as plugin crap dev4 for new directory update removed 12 old better apps around 4 51 first 6 moving 0 been after add additional sitemgr switching default identical converted incl updated catalan php remove the now index cleanups plus 52 1240675 branch fixes 50 places set 65 documentation to phpgwapi dir svn of anymore system and reverted labbe based amazon docs addressbook applying rc1 hopefully

(b) EGroupware

api stuff updates a nav page all indent warnings nuke 2005 structure e17 argh add ie split cleanup quite removed fixes html we fixed allow breaks as adding there strings bump first direcotry merging ability mon 0 dead merge doing keys bug works headers 22x22 data setting for may 28 correct head themes images forgot imenu this edb about copyrights lately seb 17 foucs icons actually copyright bit test update ones cleanups asm work eapi 2000 have anyway menus wed do added pages download autof00 browsers patch arguments hide rendering kainx also wshadow configuration ran doxygen better files hopefully move

(c) Enlightenment

zoneinfo removing translation copyright license moved revision 18 gpl between make nntp address lots directory a the because 29 except to api library compile fixing translations clean for federico updated rewrite 03 my almost

(d) Evolution

initial 1 moved const license a remove cleaning tests firebird isql interfaces vulcan removed cleanup stylesheets all due nuke imports names at are associated src ctx instructions test unneeded below backup conversion jim others message 0 free struct add c split 2 no msvc7 progress when and install arbitrary corrections files correctness handling unzipped structure chance environment fresh 3 original allocation icu been early borland gds directories dbschema backport testware intlcpp memory default 40 firebirdclient borrie new updated unmodified belong the cache cleanups builds deleted style rebench pool series local variables inprise added allocator macro 10 misc handle change fbsqlexception w3 cr on ironing integrate move netprovider refactoring import back collations don global unified cvs older after fb gateway status tabify headers finished changed s fiel fix head in assert driver merged alvarez warning forums v1 branch fixes common firebird2 65 4th dir fixed compile requiem used poor final functions related private symbol code better g trace characters another

(e) Firebird

atis config auto constants mvdir r 05 crash 98 parsing sets 13 should discless marker append afghanistan initializing vc grant 140 covarage delimiters changed engine 120 alt000001 future 1 3rd 2 bin 62 accepted 56 skip bitkeeper perl leak cmvmi build 2000 to array found 78 fixed atoll int flexbench bk updating acconfig allocated ndb strmov tools 01 symbol docs g cacert files 0009

(f) MySQL 5.0

initial 98 copyright docs gborg did create cvs documentation and cleaned some pghackers ends work anything abort head going refer tag in remaining dependencies time add support

(g) PostgreSQL

initial creating preparing removing docbook rcs 3 debian check jermey regenerate new about no need can is 18 tree header auto just shuffling through than convert pre3 been creat stuff be bulk main s character fix for updated 30 got or jeremy 2 a remove were update hellier removed then conf using access regenerated able really still compile samba but used commit 1997 attic afs based docs on if ntstatus not files andrew

(h) Samba

javadoc 0 polishing null polish rework renamed backported carriage advice reference scoping main sample 1 adapted formatting completed to code suspension aspectjautoproxycreator header initial tests commits refactoring modularization working consistency avoid first cvs log refined commons reorganized after ws primary annotation introduced consolidation different by guide abstractjdbcclinic src org phonebook evaluation tag in spring 17 review a scripting update m1 hibernate3 removed build work instrument naming repository internal method documentation cleans extended samples related jstl executor out based map binding files move

(i) Spring Framework

Figure 2. Text Clouds of Tokens used in J48 trees for each project