Boosting Automatic Commit Classification Into Maintenance Activities By Utilizing Source Code Changes

Stanislav Levin
Tel Aviv University
The Blavatnik School of Computer Science
Tel Aviv, Israel
stanisl@post.tau.ac.il

Amiram Yehudai
Tel Aviv University
The Blavatnik School of Computer Science
Tel Aviv, Israel
amiramy@tau.ac.il

ABSTRACT

Background: Understanding maintenance activities performed in a source code repository could help practitioners reduce uncertainty and improve cost-effectiveness by planning ahead and pre-allocating resources towards source code maintenance. The research community uses 3 main classification categories for maintenance activities: Corrective: fault fixing; Perfective: system improvements; Adaptive: new feature introduction. Previous work in this area has mostly concentrated on evaluating commit classification (into maintenance activities) models in the scope of a single software project.

Aims: In this work we seek to design a commit classification model capable of providing high accuracy and Kappa across different projects. In addition, we wish to compare the accuracy and kappa characteristics of classification models that utilize word frequency analysis, source code changes, and combination thereof.

Method: We suggest a novel method for automatically classifying commits into maintenance activities by utilizing source code changes (e.g, statement added, method removed, etc.). The results we report are based on studying 11 popular open source projects from various professional domains from which we had manually classified 1151 commits, over 100 from each of the studied projects. Our models were trained using 85% of the dataset, while the remaining 15% were used as a test set.

Results: Our method shows a promising accuracy of 76% and Cohen's kappa of 63% (considered "Good" in this context) for the test dataset, an improvement of over 20 percentage points, and a relative boost of ~40% in the context of cross-project classification.

Conclusions: We show that by using source code changes in combination with commit message word frequency analysis we are able to considerably boost classification quality in a project agnostic manner.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; **Maintaining software**;

KEYWORDS

Software Maintenance, Mining Software Repositories, Predictive Models, Human Factors

1 INTRODUCTION

Three main classification categories for maintenance activities in software projects were identified by Mockus et al.[1]:

- Corrective: fixing faults, functional and non-functional.
- Perfective: improving the system and its design.
- Adaptive: introducing new features into the system.

Understanding these maintenance activities, performed in a source code repository, could help practitioners reduce uncertainty and improve cost-effectiveness [2] by planning ahead and pre-allocating resources towards source code maintenance. Maintenance activity profiles of software projects have therefore been a subject of research in numerous works [1–6]. To determine maintenance activity profiles, one must first classify the activities, which come in the form of developer commits to the version control system (VCS). A widely practiced method for commit classification has been inspecting the commit's textual comment field (a.k.a commit message) [1, 7–9]. Works employing comment based classification models reported the accuracy to average below 60% when used in the scope of a single project, and below 53% when used in the scope of multiple projects (i.e., when a single model was used to classify commits from multiple projects) [9, 10]. Our work is motivated by the following observations:

(1) Existing results rarely consider cross-project classification, which threatens external validity. Hindle et al. [10] explored cross-project classification and reported the accuracy to be ~52%, which is considerably lower than the ~60% range reported by studies dealing with a scope of a single project.

(2) Existing classification results rarely report Cohen's kappa (hence forth Kappa) metric, which accounts for cases where classification categories (a.k.a classes) are unevenly distributed. Such cases make the accuracy metric somewhat misleading. For example, if the corrective class accounted for 98% of the commits, and each of the remaining classes accounted for 1% of the commits, then a simple classification model which always classified commits as corrective would have an impressive accuracy of 98%. Its Kappa on the other hand, would be 0, making this model much less appealing.

(3) Our previous work [5] shows that source code change types as defined by Fluri et al. [11] are statistically significant in the context of maintenance activity categories defined by Mockus et al. [1]. We believe that boosting (i.e. increasing)

the accuracy and kappa characteristics of commit classification into maintenance activities could improve the quality and accuracy of developers' maintenance profiles and the prediction models thereof [5] (see also section 8).

In this work we seek to design a commit classification model capable of providing high accuracy and Kappa across different projects. Our intuition is to try and capture such information, that is not unique to commits made in one project or another, but is of a rather generic nature. Fluri's taxonomy of source code changes for object-oriented programming languages (OOPLs) [11] consists of 48 (47 + an "unknown type") different change types, all of which are project agnostic and describe a meaningful action performed by a developer in a commit (e.g., *statement_delete*, *statement_insert*, *removed_class*, *additional_class* etc). Fluri's taxonomy of source code changes is therefore a nice fit for our need to capture project agnostic information pertaining to developer commits.

RQ. 1. Can source code changes be utilized to boost commit classification into maintenance activities?

RQ. 2. How does the quality of models which utilize source code changes compare to that of traditional models which use word frequency analysis?

2 RELATED WORK

Classifying commits into maintenance activities is commonly accomplished by inspecting commits' comment text and searching for indicative keywords [1, 5, 7, 8, 10]. Such keywords can be obtained using various techniques, such as a word-frequency analysis with normalization (e.g., stemming). Mockus et al. [1] was the first to employ a comment based commit classification, and reported the accuracy to be ∼60% when this method was applied in the scope of a single project - a large multi-million line real-time telecommunications software system.

Recent work explored using additional information such as commits' author and module, to classify commits both within a single software project, and cross-projects [10]. Within a single project, the reported accuracy ranged from ∼35% to 70% (accuracy fluctuated considerably depending on the project), for cross-project classification the accuracy was ∼52% [10].

A slightly different technique was used by Amor et al. [9] who explored classifying maintenance activities in the FreeBSD project by applying a Naive Bayes classifier on commits' comments without an apparent use of keywords. The reported accuracy of classifying a random sample was ∼70%. The sample's size was not specified.

A summary of the exiting results for commit classification into maintenance activities can be found in table 1.

Table 1: Classifying commits into maintenance activities, existing results [1, 9, 10]

Scope	Max Accuracy
Single Project	70%
Cross-Project	52%

In this work we were able to achieve an accuracy of 76% and Cohen's kappa of 63% in the context of cross-project commit classification, an improvement of over 20 percentage points and a relative boost of ∼40% in accuracy compared to previous results.

3 STATISTICAL METHODS

Picking the optimal classifier for a real-world classification problem is hardly a simple task [12], however, Random Forest (RF) [13, 14] and Gradient Boosting Machine (GBM) [15–19] classifiers are generally considered top performers [12, 18]. In addition, we also use J48 [20, 21], a variation of the C4.5 [22] algorithm. The RF and GBM models are most likely to outperform the simpler J48, but the latter, in contrast to the formers, is capable of providing a human readable representation of its decision tree. We find this ability valuable, since inspecting the decision tree can provide additional insights. An example of a decision tree produced by the J48 classifier can be found in figure 1. It is a decision tree for our keyword based commit classification model, described in section 6.

To evaluate the different commit classification models we employ common statistical measures for classification performance. For a given class $L \in \{adaptive, corrective, perfective\}$, TP_L is the number of commits correctly classified as class L; FP_L is the number of commits incorrectly classified as class L; FN_L is the number of commits of class L that were incorrectly classified.

- *Precision$_L$* = $\frac{TP_L}{TP_L + FP_L}$, the number of commits correctly classified as class L, divided by the total number of commits classified as class L.

- *Recall$_L$* = $\frac{TP_L}{TP_L + FN_L}$, the number of commits correctly classified as class L, divided by the actual number of L class commits in the dataset.

- *Accuracy* = $\frac{\sum_{L \in \{a,c,p\}} TP_L}{\sum_{L \in \{a,c,p\}} (TP_L + FP_L)}$, the proportion of correctly classified commits out of all classified commits.

- *No Information Rate (NIR)*, measures the accuracy of a trivial classifier which classifies all commits with using a single class, the one that is most frequent, in our case - corrective.

- *Kappa* - Cohen's kappa, often considered helpful as a measure that can handle both multi-class and imbalanced class problems (see section 1).

- *P-Value [Accuracy > NIR]*, the *p*-value for the null hypothesis that the "Accuracy ≤ NIR" (i.e., the accuracy of a given predictive model) . A low *p*-value allows one to reject the null hypothesis in favor of the alternative hypothesis that the "Accuracy > NIR".

4 RESEARCH METHOD

Our work consists of the following main procedures:

(1) Select candidate software projects and harvest their commit data such as commit message and source code changes performed as part of the commit (see sections 5.1, 5.2).

(2) Create a labeled commit dataset by sampling commits and manually labeling them. Each label is a maintenance category, i.e. one of the following: corrective, perfective, or adaptive (see section 5.3).

(3) Inspect the agreement level of the manual labeling procedure by sampling 10% of the labeled dataset created by the first author, and have the second author independently label it (see section 5.4).

(4) Devise predictive models that utilize source code changes for the task of commit classification into maintenance activites (see section 6).

(5) Evaluate the devised models using two mutually exclusive datasets obtained by splitting the labeled dataset into (1) a *training* dataset, consisting of 85% of the labeled dataset, and (2) a *test* dataset, consisting of the remaining 15% of the labeled dataset which was never employed as part of the training process (see section 7).

5 DATA COLLECTION

5.1 Selecting candidate software projects

We use GitHub [23] as the data source for this work due to its popularity and rich query options. Candidate repositories were selected according to the following criteria, which we designed to target data-rich repositories:

- Used the Java programming language
- Had more than 100 stars (i.e. more than 100 users have "liked" these repositories)
- Had more than 60 forks (i.e., more than 60 users have "copied" these repositories for their own use)
- Had their code updated since 2016-01-01 (i.e., these repositories are active)
- Were created before 2015-01-01 (i.e., these repositories have been around)
- Had size over 2MB (i.e. these repositories are of considerable size)

Out of all candidates we selected 11 projects which are well known in the open source community, and cover a wide range of software domains such as IDEs, programming languages (that were implemented in Java), distributed database and storage platforms, and integration frameworks.

(1) **RxJava** - a library for composing asynchronous and event-based programs for the Java VM.
(2) **Intellij Community Edition** - A popular IDE for the Java programming language.
(3) **HBase** - A distributed, scalable, big data store.
(4) **Drools** - A Business Rules Management System solution.
(5) **Kotlin** - A Statically typed programming language for the JVM, Android and the browser by JetBrains.
(6) **Hadoop** - A framework that allows for the distributed processing of large data sets across clusters of computers.
(7) **Elasticsearch** - A distributed search and analytics engine.
(8) **Restlet** - A RESTful web API framework for Java.
(9) **OrientDB** - A Distributed Graph Database with the flexibility of Documents in one product.
(10) **Camel** - An open source integration framework based on known Enterprise Integration Patterns.
(11) **Spring Framework** - An application framework and inversion of control container for the Java platform.

5.2 Distilling source code changes

The source code changes distilling was carried out using a designated VCS mining platform we have built on top of Spark [24, 25], a state of the art framework for large data processing.

After downloading (cloning) the repositories from GitHub, for each repository r where $1 \leq r \leq 11$ we created a series of patch files $\{p_i^r\}_{i=1}^{N_r}$, where N_r is the latest revision number for repository r. Each patch file p_i^r was responsible for transforming repository r from revision r_{i-1} to revision r_i, where r_0 is the empty repository. By initially setting repository r to revision 1 (i.e. the initial revision) and then applying all patches $\{p_i^r\}_{i=2}^{N_r}$ in a sequential manner, the revision history for that repository was essentially replayed. Conceptually, this was equivalent to the case of all developers performing their commits sequentially one by one according to their chronological order.

To distill source code changes as per the taxonomy defined by Fluri et al., we repeatedly applied ChangeDistiller ([26–29]) on every two consecutive revisions of every Java file in every repository we had selected to be part of the dataset, essentially extracting source code changes from the entire project's history.

5.3 Creating a labeled commit dataset

The first author *manually* classified a randomly sampled set of ∼100 commits from each of the studied 11 repositories. To improve classification quality the projects' issue tracking systems (e.g. JIRA by Atlassian [30]) was often used. The JIRA contained the tickets occasionally referenced in developers' commits. Such tickets (a.k.a. issues) typically contain additional information about the feature or bug the referencing commit was trying to address. Moreover, tickets sometimes had their own classification categories such as "feature request", "bug", "improvement" etc., but unfortunately they were not very reliable as developers were not always consistent with their categories. For instance, in some cases bug fixes were labeled as "improvement", and while fixing a bug is indeed an improvement, according to the classification categories we use (Mocuks et. al. [1]), bug fixes should be classified corrective while improvements should be classified perfective. Some developers used the term "fix" even when they referenced feature requests, e.g. "fixed issue #N", where "issue #N" spoke of a new feature or an improvement that did not necessarily report a bug. These observations are consistent with Herzig et al [31] who reported that 33.8% of the bug reports they studied were misclassified.

In cases where the lack of supporting information (e.g., indescriptive ticket and / or commit message) prevented us from classifying a certain commit with satisfactory confidence, that commit was dropped from the dataset and replaced by a new one, selected randomly from the same project repository. If we were unable to classify the replacement commit as well, we would repeat this routine until we found a commit that we were able to confidently classify. Further rules of thumb we used for classifying were as follows:

- Javadoc and comment updates were considered perfective
- Fixing a broken unit test or build was considered corrective
- Adding new unit test(s) was considered perfective
- Performance improvements that resulted from an open ticket in the issue tracking system were considered corrective

- Performance improvements that did NOT result from an open ticket in the issue tracking system were considered perfective

We made efforts to prevent class starvation (i.e., not having enough instances of a certain class) which could in turn substantially degrade models' performance, and in case we detected a considerable imbalance in some project's classification categories (a.k.a classes), we added more commits of the starved class from the same project. This balancing was done by repeatedly sampling and manually classifying commits until a commit of the starved class was found. The final dataset consisted of 1151 manually classified commits and was made open access [32]. This dataset contained 100-115 from each project. 43.4% (500 instances) were corrective, 35% (404 instances) were perfective, and 21.4% (247 instances) were adaptive. These commits yielded 33,149 source code changes.

5.4 Inspecting manual labeling agreement

In order to inspect manual classification agreement, we randomly selected 110 commits out of the 1151 commits that had been labeled by the first author, 10 random commits from each of the 11 projects, and had these commits independently labeled again by the second author.

At first the agreement stood at 79%. After discussing the conflicts and sharing the guidelines used by the first author in more detail, the agreement level rose to 94.5%. According to the one sample proportion test [33], the error margin for our observed agreement level was 4.2%, and the estimated asymptotic 95% confidence interval was [90.3%, 98.7%]. This indicates that both authors were in agreement about the labels for the vast majority of cases once they employed the same guidelines (see section 5.3).

6 COMMIT CLASSIFICATION MODELS

We split the labeled dataset into a training dataset and a test dataset, 85% and 15% respectively. The model training phase consists of using 5 time repeated 10-fold validation for each compound model (which boils down to performing a 10-fold cross validation process 5 different times and averaging the results). Then, the trained models are evaluated using the test dataset - the 15% split that did not take part in the model training process. Our statistical computations are carried out in the R statistical environment [34], where we extensively use the R caret package [35] for the purpose of model training and evaluation.

6.1 Utilizing word frequency analysis

First we classified the test dataset (the 15% of the entire labeled dataset) using a naive method to set an initial baseline. The naive method is based solely on searching for pre-defined words gathered from previous work [5], and returning the most frequent class (i.e., corrective) in case none of the keywords were present in a commit's message, see table 2 for more details. The results showed that 34.8% of the commits in the test dataset (60 commits) did not have any of the keywords present in their commit message, and were therefore automatically classified corrective. In addition, we can note the low recall of the perfective class, as opposed to the high recall of the corrective class (which accounts for most of the commits in the classified dataset).

Table 2: Naive model's confusion matrix

true class / classified as	adaptive	corrective	perfective
adaptive	**18**	2	16
corrective	18	**72**	37
perfective	1	1	**7**
Recall:	48%	96%	11%
Precision:	50%	56%	77%
Accuracy:	56%		
Kappa:	29%		
No Information Rate (NIR):	43%		
P-Value [Accuracy > NIR]:	0.0005		

Due to the high percentage of commits without any of the keywords we had defined, we then tried to fine-tune the keywords we search for. We performed an additional experiment using the same classification method, only this time the keywords were obtained by employing a word frequency analysis with normalization of the commit messages. This time 28% of the commits did not have any of the keywords present in their commit message. These findings led us to believe that the number of commits having none of the keywords used by keyword based classification models is quite considerable.

6.2 Utilizing source code changes (RQ. 1)

The subject of dealing with missing values in a classification problem is broadly covered by Saar-Tsechansky et al. [36], who describe two common methods employed to overcome this issue: (1) imputation, where the missing values are estimated from the data that are present, and (2) reduced-feature models, which employ only those features that will be known for a particular test case (i.e., only a subset of the features that are available for the entire training dataset), so that imputation is not necessary. Since our dataset consists of two different data types, keywords and source code changes, we use reduce-feature models, which are reported to outperform imputation [36] and represent our use-case more naturally. In addition, since the missing feature patterns in our dataset are known in advance, i.e., given a commit only the keywords can be missing, its source code changes are always present, we can pre-compute and store two models; one to be used when all features are present (keywords + source code changes), and the other when only a subset is available (source code changes only). We define the notion of a **compound** model (similarly to the "classifier lattice" [36]) which uses two separate models for classifying commits with, and without (pre-defined) keywords in their commit message. The classify routine of the compound model is pseudo-coded in listing 1.

Listing 1: Compound model's classify routine

```
classify(commit) {
    if(hasKeywords(commit.comment)) {
        classifyWith(model_KW, commit);
    } else {
        classify(model_KW_bar, commit);
    }
}
```

Given a commit C, the compound model first checks if C's commit message has any keywords, if so, the model defined as model_{KW} is used to classify C, otherwise (i.e., no keywords found in C's commit message), the model defined as $\text{model}_{\overline{KW}}$ is used to classify C. Each of the models model_{KW} and $\text{model}_{\overline{KW}}$ may or may not be a reduced-feature model, depending on whether it employs the full set of features (both keywords and source code changes), or only a subset of it (either keywords or source code changes).

We define $\text{model}_{\overline{KW}}$ and model_{KW} to be one of the following model types:

- **Keywords** model, which relies solely on keywords to classify commits. The features used by this model are keywords obtained by performing the following transformations on the commit comment fields:
  (1) Removed special characters
  (2) Made lower case (case-folding)
  (3) Removed English stopwords
  (4) Removed punctuation
  (5) Striped white-spaces
  (6) Performed stemming
  (7) Adjusted frequencies so that each comment can contribute a given word only once
  (8) Removed custom words such as developer names, projects names, VCSs lingo (e.g., head, patch, svn, trunk, commit), domain specific terms (e.g., http, node, client): "patch", "hbase", "checksum", "code", "version", "byte", "data", "hfile", "region", "schedul", "singl", "can", "yarn", "contribut", "commit", "merg", "make", "trunk", "hadoop", "svn", "ignoreancestri", "node", "also", "client", "hdfs", "mapreduc", "lipcon", "idea", "common", "file", "ideadev", "plugin", "project", "modul", "find", "border", "addit", "changeutilencod", "clickabl", "color", "column", "cach", "jbrule", "drool", "coprocessor", "regionserv", "scan", "resourcemanag", "cherri", "gong", "ryza", "sandi", "xuan", "token", "contain", "shen", "todd", "zhiji", "tan", "wangda", "timelin", "app", "kasha", "kashacherri", "messag", "spr", "camel", "http", "now", "class", "default", "pick", "via".
  (9) We then selected the 10 most frequent words from each of the three maintenance activities in the test dataset:
    – Corrective: (1) *fix* (2) *test* (3) *issu* (4) *use* (5) *fail* (6) *bug* (7) *report* (8) *set* (9) *error* (10) *npe*
    – Perfective: (1) *test* (2) *remov* (3) *use* (4) *fix* (5) *refactor* (6) *method* (7) *chang* (8) *add* (9) *improv* (10) *new*
    – Adaptive: (1) *support* (2) *add* (3) *implement* (4) *new* (5) *allow* (6) *use* (7) *method* (8) *test* (9) *set* (10) *chang*
  It can be seen that some of the words (as obtained by our commit message word frequency analysis) overlap

between categories. The words *"test"* and *"use"* appear in all three categories; the word *"fix"* appears in both the corrective and perfective categories; the words *"method"*, *"chang"*, *"add"* and *"new"* appear both in the perfective and adaptive categories; and the word *"set"* appears both in the corrective and adaptive categories. These word overlaps may indicate that keywords alone are insufficient to accurately classify commits into maintenance activities, and need to be augmented with additional information in order to improve classification accuracy.

For the purpose of building the Keywords model type, we remove multiple occurrences of the same word (so that each word appears only once per maintenance category) and remain with the following set of words: (1) add (2) allow (3) bug (4) chang (5) error (6) fail (7) fix (8) implement (9) improv (10) issu (11) method (12) new (13) npe (14) refactor (15) remov (16) report (17) set (18) support (19) test (20) use.

- (Source Code) **Changes** based model, which relies solely on source code changes to classify commits. The features used by this model are source code change types [11] obtained by distilling commits, as described earlier in this section.
- **Combined** (Keyword + Source Code Change Types) model, which uses both keywords and source code change types to classify commits. The features used by this type of models consist of both keywords and source code change types.

A summary of the model components can be found in table 3.

Table 3: Reduced-feature model components

Model Type	Model Features
Keywords	Words
Changes	Source Code Change Types
Combined	Words + Source Code Change Types

For example, a commit where two methods were added (source code change type *"additional_functionality"*), and one statement was updated (source code change type *"statement_updated"*) and has a commit message that says *"Refactored blob logic into separate methods"* will be treated differently by each of the model types indicated in table 3.

The Keywords model extracts features represented by tuples of size 20, and given the commit above would extract the following features: $(\overbrace{0\ldots1\ldots1\ldots0}^{20})$ with "1" in the coordinates that represent the words *"refactor"* and *"method"*. The count of each keyword is at most one, i.e., duplicate keywords are counted only once. Source code changes are ignored, since the Keywords model type does not consider source code changes.

The Changes model extracts features represented by tuples of size 48 (since there are 48 different source code change types), and given the commit above would extract the following features: $(\overbrace{0\ldots2\ldots1\ldots0}^{48})$ with "2" in the coordinate that represents the

source code change type *"additional_functionality"* and "1" in the coordinate that represents *"statement_updated"*. In contrast to the case of the Keywords model, all occurrences of every source code change type are counted in. Keywords in the commit message are ignored, since the Changes model type does not consider keywords. The Combined model extracts features represented by tuples of size 68 (= 48 source code change types + 20 keywords), and given the commit above would extract the following features: $(\overbrace{0\ldots1\ldots1\ldots0}^{68}\underbrace{\ldots0\ldots2\ldots1\ldots0}_{})$, with "2" in the coordinate that represents the source code change type *"additional_functionality"*, and "1" in the coordinates that represent the source code change type *"statement_updated"*, the keyword *"refactor"*, and the keyword *"method"*. The Combined model type captures both keywords and source code change types - hence its name.

In the next sections we evaluate and compare different compound models by considering the different combinations of their model$_{KW}$ and model$_{\overline{KW}}$ model components. The evaluation process consists of the following steps:

(1) Select the model component model$_{KW}$
(2) Select the model component model$_{\overline{KW}}$
(3) Select an underlying classification algorithm for the compound model, which determines the algorithm to be used by each of the model components Model$_{KW}$ and Model$_{\overline{KW}}$ (J48, GBM, or RF see section 3).

7 RESULTS (RQ. 2)

Table 4 describes an exhaustive set of combinations for selecting the pair of (Model$_{KW}$, Model$_{\overline{KW}}$) models, given that each can be one of the three model types defined in table 3. Each row in table 4 represents a compound model, defined by the selection of (Model$_{KW}$, Model$_{\overline{KW}}$). The classification accuracy and Kappa achieved by a given compound model are reported in the corresponding Accuracy and Kappa columns. The best performing compound model for each classification algorithm is highlighted in lime-green, and the keywords based model (where both Model$_{KW}$ and , Model$_{\overline{KW}}$ are of the Keywords model type) is highlighted in orange so that it can be easily compared to compound models that utilize source code changes.

Following our main research questions (see section 1), the accuracy and Kappa results for each compound model during the training (see table 4) reveal that the compound models that use either Model$_{\overline{KW}}$ = Combined or Model$_{\overline{KW}}$ = Changes achieve higher accuracy and Kappa when compared to models with the same Model$_{KW}$ component but with Model$_{\overline{KW}}$ = Keywords, regardless of the underlying classification algorithm (J48, GBM or RF). This comes as no surprise, as one could expect keyword based models would have trouble accurately classifying commits that do not have any keywords in their commit message. Table 4 also reveals that models that rely solely on commit messages have higher accuracy and kappa than models that rely solely on source code changes (under all three algorithms).

Table 4: Training dataset compound models performance

Alg.	Model$_{KW}$	Model$_{\overline{KW}}$	Accuracy	Kappa
J48	Combined		69.0%	51.7%
	Combined	Keywords	67.7%	50.2%
	Combined	Changes	69.2%	51.9%
	Keywords	Combined	69.8%	53%
	Keywords		68.5%	51.5%
	Keywords	Changes	69.9%	53.2%
	Changes	Combined	48.7%	20.1%
	Changes	Keywords	47.4%	17.2%
	Changes		48.8%	18.6%
GBM	Combined		72.0%	56.2%
	Combined	Keywords	69.0%	51.8%
	Combined	Changes	72.0%	55.9%
	Keywords	Combined	71.6%	56.0%
	Keywords		68.5%	51.4%
	Keywords	Changes	71.5%	55.6
	Changes	Combined	54.1%	26.9%
	Changes	Keywords	51.0%	22.4%
	Changes		54.3%	26.9%
RF	Combined		73.1%	57.8%
	Combined	Keywords	69.5%	52.6%
	Combined	Changes	71.9%	55.7%
	Keywords	Changes	72.2%	56.4%
	Keywords	Combined	73.6%	58.9%
	Keywords		69.8%	53.4%
	Changes	Combined	54.5%	26.6%
	Changes	Keywords	50.6%	21.1%
	Changes		52.9%	23.4%

Table 5: Training dataset accuracy, best model per algorithm

Alg.	Min.	1-st Q.	Median	Mean	3-rd Q.	Max.
J48	60.8%	66.4%	70.1%	69.9%	73.4%	80.6%
GBM	60.8%	69.2%	72.1%	72.0%	75.2%	80.8%
RF	65.6%	70.4%	73.4%	73.6%	76.6%	82.8%

Further accuracy and Kappa statistics pertaining to the training stage of the best performing model for each algorithm can be found in table 5 and table 6 respectively. From table 5 and table 6 we can learn that during the training stage, the RF model consistently outperforms the J48 and even the GBM model, in both accuracy and Kappa, across all of the cuts: minimum, 1-st quartile (25-th percentile), median, mean, 3-rd quartile (75-th percentile) and maximum. In particular, the minimum accuracy and Kappa of the RF are notably higher than its competitors.

Table 6: Training dataset Kappa, best model per algorithm

Alg.	Min.	1-st Q.	Median	Mean	3-rd Q.	Max.
J48	38.4%	47.9%	53.4%	53.2%	58.8%	69.7%
GBM	38.3%	51.8%	56.9%	56.2%	60.6%	70.0%
RF	45.5%	54.1%	58.6%	58.9%	63.3%	73.5%

The top performing models were then used to classify the test dataset, consisting of 15% of the entire labeled dataset, see table 7. The ultimate winner was the RandomForest compound model with Model_{KW} = Keywords and $\text{Model}_{\overline{KW}}$ = Combined. A detailed confusion matrix for this champion model can be found in table 8.

Table 7: Test dataset classification performance

Algorithm	Model_{KW}	$\text{Model}_{\overline{KW}}$	Accuracy	Kappa
J48	Keywords	Changes	70.3%	53.9%
GBM	Combined		72.6%	57.2%
RF	Keywords	Combined	76.7%	63.5%

The decision tree built by the J48 algorithm for our keyword based model (see figure 1) provides some interesting insights regarding its classification process. The word "fix" is the single most indicative word of corrective commits, which aligns well with our intuition, according to which commits that fix faults are likely to include the "fix" noun or verb in the commit message. Given that "fix" did not appear, the words "support" and "allow" are most indicative of adaptive commits, presumably these words are used by developers to indicate the support of a new feature, or the fact that something new is now "allowed" in the system. The combination "implement chang" (stemmed), given that "fix", "support" and "allow" did not appear, is very indicative of either perfective or corrective commits, if however, "implement" is not accompanied by the word "chang" (stemmed), the commit is likely to be adaptive. The (stemmed) word "remov", given that the words "fix", "support", "allow" and "implement" did not appear, is very indicative of perfective commits, perhaps because developers often use it to describe a

Table 8: Keywords-Combined RF compound model's confusion matrix for the test dataset

classified as \\ true class	adaptive	corrective	perfective
adaptive	**28**	5	5
corrective	6	**63**	14
perfective	3	7	**41**
Recall:	75%	84%	68%
Precision:	73%	75%	80%
Accuracy:	76%		
Kappa:	63%		
No Information Rate (NIR):	43%		
P-Value [Accuracy > NIR]:	$< 2e^{-16}$		

modification where they remove an obsolete mechanism in favor of a new one.

We also visualized the maintenance categories keyword (see section 6.2) frequency using a word-cloud [1], which revealed that the word "test" is particularly common in perfective commits, but is generally common in all three maintenance activity types. The word "use" is also common in all three maintenance activity types, but is particularly frequent in the perfective category. The words "fix", "remov" and "support" are quite distinctive of their corresponding maintenance activity types: corrective, perfective and adaptive categories (respectively). The word "add" is common in adaptive commits, as well as "allow".

Similarly, the source code changes frequencies can also be visualized using a source-code-change-type-cloud [2] which reveals that statement related changes, e.g., "statement_insert", "statement_update" and "statement_delete" are the most common change types in all three maintenance activities (corrective, perfective, adaptive). The semantic change type "additional_functionality" is common in both perfective and adaptive commits, but less so in corrective commits.

The term-cloud and J48 keyword based decision tree visualizations provide an intuition for why J48 is likely to outperform a simple word-frequency based classification. In contrast to the word-cloud, which provides "flat" frequencies, the J48 is capable of capturing information pertaining to the presence of multiple keywords in the same commit message, as indicated by the decision tree. In addition, the predictor importance analysis for the champion RF model (omitted for brevity), shows numerous change types rank high, confirming their viability for the classification process.

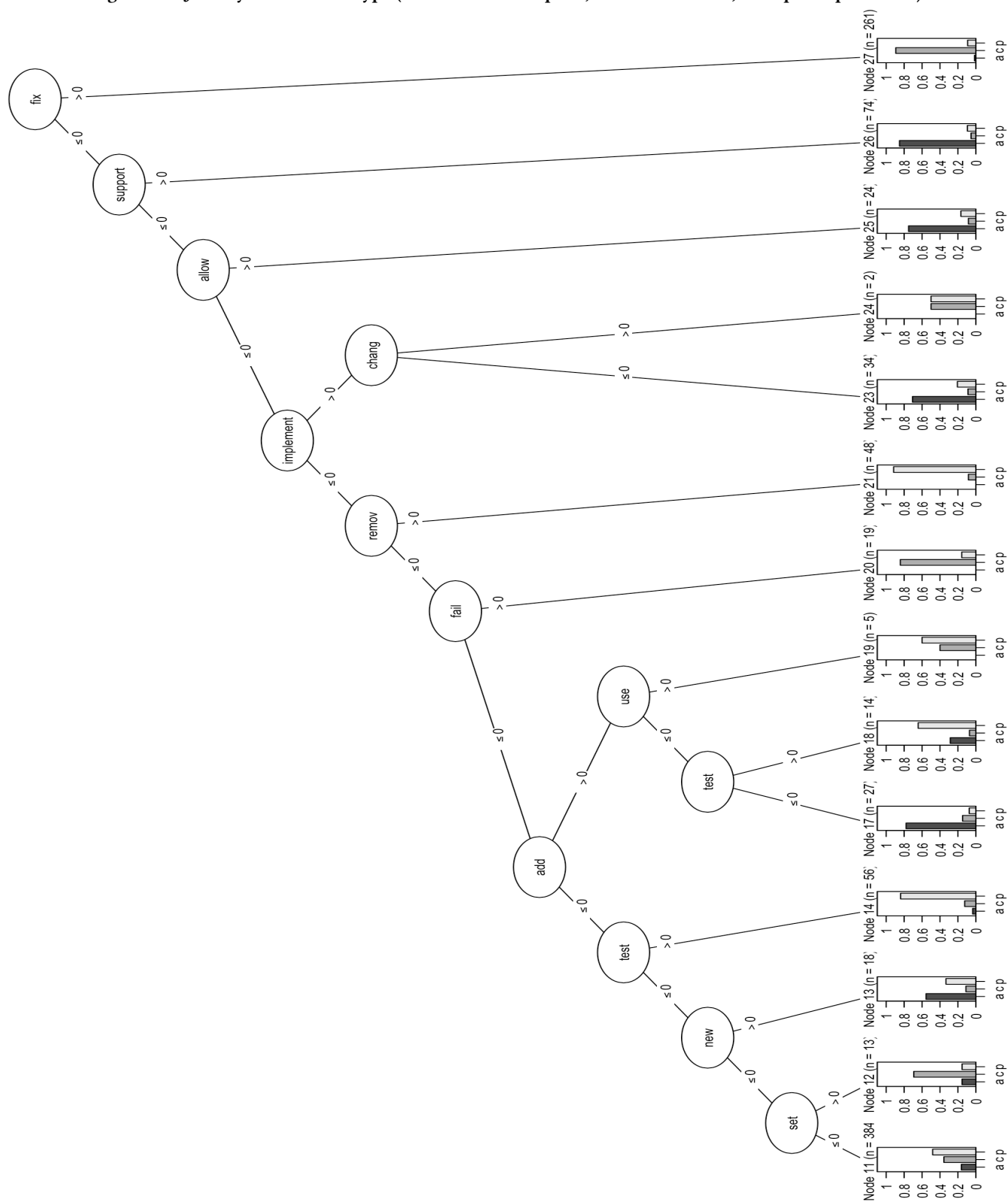## 8 DISCUSSION & APPLICATIONS

Improving developer's maintenance profile accuracy. Our previous work [5] suggested the notion of a developer's maintenance profile, which describes the amount of commits a given developer made in each of the maintenance categories (corrective, perfective, adaptive). The models we devised in order to predict developers maintenance profiles could benefit from a more accurate classification of commits into maintenance activities as part of their training stage, possibly yielding higher prediction quality.

Identifying anomalies in development process. The manager of a large software project should aim to control and manage its maintenance activity profiles, i.e., the volume of commits made in each maintenance activity. Monitoring for unexpected spikes in maintenance activity profiles and investigating the reasons (root cause) behind them would assist managers and other stakeholders to plan ahead and identify areas that require additional resource allocation. For example, lower corrective profiles could imply that developers are neglecting bug fixing. Higher corrective profiles could imply an excessive bug count. Finding the root cause in cases

---

[1]https://github.com/staslev/paper-resources/raw/promisedata-2017/Boosting-Automatic-Commit-Classification-Into-Maintenance-Activities-By-Utilizing-Source-Code-Changes/word-cloud.png

[2]https://raw.githubusercontent.com/staslev/paper-resources/promisedata-2017/Boosting-Automatic-Commit-Classification-Into-Maintenance-Activities-By-Utilizing-Source-Code-Changes/change-cloud.png

Figure 1: A J48 Keywords model type ("a" stands for adaptive, "c" for corrective, and "p" for perfective)

of significant deviations from predicted values may reveal essential issues whose removal can improve projects' health. Similarly, exceptionally well performing projects can also be a good subject for investigation in order to identify positive patterns.

Improving development team's composition. Building a successful software team is hardly a trivial task as it involves a delicate balance between technological and human aspects [37, 38]. We believe that by using commit classification it would be possible to build reliable developer maintenance activity profiles [5] which could assist in composing balanced teams. We conjecture that composing a team that heavily favors a particular maintenance activity (e.g. adaptive) over the others could lead to an unbalanced development process and adversely affect the team's ability to meet typical requirements such as developing a sustainable number of product features, adhering to quality standards, and minimizing technical debt so as to facilitate future changes.

9 THREATS TO VALIDITY

Threats to Statistical Conclusion Validity are the degree to which conclusions about the relationship among variables based on the data are reasonable.

Our results are based on manually classifying 1151 commits, over 100 commits from each of the studied 11 projects. The projects originated from various professional domains such as IDEs, programming languages, distributed database and storage platforms, and integration frameworks. Each compound model was trained using 5-time repeated 10-fold cross validation. In addition, our commit classifications evaluations demonstrated p-value below 0.01, supporting the statistical validity of the hypothesis accuracy > NIR with high confidence.

Threats to Construct Validity consider the relationship between theory and observation, in case the measured variables do not measure the actual factors.

- Manual Commit Classification. We took the following measures to mitigate manual classification related errors:
  (1) Projects' issue tracking systems were used, and often provided additional information pertaining to commits.
  (2) Commits that did not lend themselves to classification due to lack of supporting information were removed from the dataset and replaced by other commits from the same repository.
  (3) Both authors independently classified 10% of the commits in the dataset used in this work. The observed agreement level was 94.5%, and the asymptotic 95% confidence interval for the agreement level was [90.3%, 98.7%] indicating that both authors agreed about the labels for the vast majority of cases.
- Source Code Change Extraction. ChangeDistiller and the VCS mining platform we have built and used are both software programs, and as such, are not immune to bugs which could result in inaccurate or incomplete source code change extraction.

Threats to External Validity consider the generalization of our findings.

- Programming Language Bias. All analyzed commits were in the Java programming language. It is possible that developers who use other programming languages, have different maintenance activity patterns which have not been explored in the scope of this work.
- Open Source Bias / GitHub. The repositories studied in this paper were all popular open source projects from GitHub, selected according to the criteria described in section 5.1. It may be the case that developers' maintenance activity profiles are different in an open source environment when compared to other environments.
- Popularity Bias. We intentionally selected the popular, data rich repositories. This could limit our results to developers and repositories of high popularity, and potentially skew the perspective on characteristics found only in less popular repositories and their developers.
- Limited Information Bias. The entire dataset, both the training and the test datasets, contained only those commits that we were able to manually classify. At the stage of VCS inspection it can be essentially impossible to actually ascertain the maintenance categories of commits that do not provide enough information traces (comment, ticket id, etc.). The true maintenance category for such commits may only be known to the developers who made them, and even they may no longer recall it soon after they have moved on to their next task.
- Mixed Commits. Recent studies [39, 40] report that commits may involve more than one type of maintenance activity, e.g. a commit that both fixes a bug, and adds a new feature. Our classification method does not currently account for such cases, but this is definitely an interesting direction to be considered for future work (see section 10).

10 CONCLUSIONS AND FUTURE WORK

We suggested a novel method for classifying commits into maintenance activities and used it to devise (and evaluate) a number of models that utilize commit message word frequency analysis and source code change extraction for the purpose of cross-project commit classification into maintenance activities. These models were then evaluated and compared using the accuracy and Kappa metrics with different underlying classification algorithms. Our champion model showed a promising accuracy of 76% and Kappa of 63% when applied on the test dataset which consisted of 172 commits originating from various projects. These results show an improvement of over 20 percentage points, and a relative boost of over 40% when compared to previous results (see table 1), see also table 2 vs. table 8 which depict the commonly used classifier and our champion classifier, respectively. Our work is based on studying 11 popular open source projects from various professional domains, from which we manually classified 1151 commits, ~100 from each of the studied projects. The suggested models were trained using repeated cross validation on 85% of the dataset, and the remaining 15% of the dataset were used as a test set.

We conclude that the answer to RQ 1. is that source code changes can indeed be successfully used to devise high quality models for commit classification into maintenance activities. The answer to

RQ 2. is that models that utilize source code changes are capable of outperforming the reported accuracy of word frequency based models ([9, 10]) from ∼60% to ∼75%, even when classifying cross-project commits.

In addition, we make the following observations based on our study:

- Using text cleaning and normalization, our word frequency based models were able to achieve an accuracy of 68-69% with Kappa of 51-53% for cross-project commits classification (see table 4).
- Compound models employing both (commit message) word frequency analysis and source code change types for the task of cross-project commit classification were able to achieve up to 73% accuracy with Kappa 59% during the training stage, and up to 76% accuracy with Kappa of 63% (considered "Good" [33]) for the test dataset.
- The RF algorithm outperformed the GBM and J48 in classifying cross-project commits (see table 7 and 8).

Having an accurate classification model and the ability to apply it at scale, it may be possible to automatically classify an unprecedentedly large number of projects and commit activities. This could facilitate the revisiting of the subject of maintenance activity distribution in software projects [4, 6].

It could also be beneficial to explore whether mixed commits ([39, 40]) could be automatically and accurately classified into hybrid categories, e.g. corrective+perfective to indicate that a given commit improves code's structure in addition to fixing a bug.

We believe that employing source code changes to help answer these questions may lead to useful insights for both practitioners and the research community.

REFERENCES

[1] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Software Maintenance, 2000. Proceedings. International Conference on.* IEEE, 2000, pp. 120–130.
[2] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd international conference on Software engineering.* IEEE Computer Society Press, 1976, pp. 492–497.
[3] W. Meyers, "Interview with wilma osborne," *IEEE Software*, vol. 5, no. 3, pp. 104–105, 1988.
[4] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Communications of the ACM*, vol. 21, no. 6, pp. 466–471, 1978.
[5] S. Levin and A. Yehudai, "Using temporal and semantic developer-level information to predict maintenance activity profiles," in *Proc. ICSME.* IEEE, 2016, pp. 463–468.
[6] S. R. Schach, B. Jin, L. Yu, G. Z. Heller, and J. Offutt, "Determining the distribution of maintenance categories: Survey versus measurement," *Empirical Software Engineering*, vol. 8, no. 4, pp. 351–365, 2003.
[7] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on.* IEEE, 2003, pp. 23–32.
[8] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.
[9] J. J. Amor, G. Robles, J. M. Gonzalez-Barahona, and A. Navarro, "Discriminating development activities in versioning systems: A case study," in *Proceedings PROMISE.* Citeseer, 2006.
[10] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt, "Automatic classification of large changes into maintenance categories," in *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on.* IEEE, 2009, pp. 30–39.
[11] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on.* IEEE, 2006, pp. 35–45.
[12] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, "Do we need hundreds of classifiers to solve real world classification problems," *J. Mach. Learn.*

[13] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE transactions on pattern analysis and machine intelligence*, vol. 20, no. 8, pp. 832–844, 1998.
[14] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.
[15] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.
[16] "R gbm package," https://cran.r-project.org/web/packages/gbm/gbm.pdf, [Online; accessed Nov-2016].
[17] G. Ridgeway, "Generalized boosted models: A guide to the gbm package," *Update*, vol. 1, no. 1, p. 2007, 2007.
[18] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in *Proceedings of the 23rd international conference on Machine learning.* ACM, 2006, pp. 161–168.
[19] R. Caruana, N. Karampatziakis, and A. Yessenalina, "An empirical evaluation of supervised learning in high dimensions," in *Proceedings of the 25th international conference on Machine learning.* ACM, 2008, pp. 96–103.
[20] E. Frank, M. Hall, G. Holmes, R. Kirkby, B. Pfahringer, I. H. Witten, and L. Trigg, "Weka," in *Data Mining and Knowledge Discovery Handbook.* Springer, 2005, pp. 1305–1314.
[21] "R/weka interface," https://cran.r-project.org/web/packages/RWeka/index.html, [Online; accessed Nov-2016].
[22] J. R. Quinlan, *C4. 5: programs for machine learning.* Elsevier, 2014.
[23] "Github - the largest open source community in the world," https://github.com/open-source, [Online; accessed 18-April-2016].
[24] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.
[25] "Lightning-fast cluster computing," http://spark.apache.org/, [Online; accessed 11-April-2016].
[26] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Software*, vol. 26, no. 1, p. 26, 2009.
[27] B. Fluri, E. Giger, and H. C. Gall, "Discovering patterns of change types," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on.* IEEE, 2008, pp. 463–466.
[28] M. Martinez, L. Duchien, and M. Monperrus, "Automatically extracting instances of code change patterns with ast analysis," *arXiv preprint arXiv:1309.3730*, 2013.
[29] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *Software Engineering, IEEE Transactions on*, vol. 33, no. 11, pp. 725–743, 2007.
[30] Atlassian, "The #1 software development tool used by agile teams," https://www.atlassian.com/software/jira, [Online; accessed 20-Mar-2017].
[31] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *Proceedings of the 2013 International Conference on Software Engineering.* IEEE Press, 2013, pp. 392–401.
[32] S. Levin and A. Yehudai, "1151 commits with software maintenance activity labels (corrective,perfective,adaptive)," Jul. 2017. [Online]. Available: https://doi.org/10.5281/zenodo.835534
[33] D. G. Altman, *Practical statistics for medical research.* CRC press, 1990.
[34] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2008, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org
[35] "The caret package," http://topepo.github.io/caret/index.html, https://cran.r-project.org/web/packages/caret/caret.pdf, [Online; accessed Nov-2016].
[36] M. Saar-Tsechansky and F. Provost, "Handling missing values when applying classification models," *Journal of machine learning research*, vol. 8, no. Jul, pp. 1623–1657, 2007.
[37] N. Gorla and Y. W. Lam, "Who should work with whom?: building effective software project teams," *Communications of the ACM*, vol. 47, no. 6, pp. 79–82, 2004.
[38] P. J. Guinan, J. G. Cooprider, and S. Faraj, "Enabling software development team performance during requirements definition: A behavioral versus technical approach," *Information Systems Research*, vol. 9, no. 2, pp. 101–125, 1998.
[39] H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on.* IEEE, 2013, pp. 138–147.
[40] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! are you committing tangled changes?" in *Proceedings of the 22nd International Conference on Program Comprehension.* ACM, 2014, pp. 262–265.