

# Improve Classification of Commits Maintenance Activities with Quantitative Changes in Source Code

Richard V. R. Mariano<sup>1</sup>, Geanderson E. dos Santos<sup>2</sup> and Wladimir Cardoso Brandão<sup>1</sup>

<sup>1</sup>Department of Computer Science, Pontifical Catholic University of Minas Gerais (PUC Minas), Belo Horizonte, Brazil

<sup>2</sup>Department of Computer Science, Federal University of Minas Gerais (UFMG), Belo Horizonte, Brazil

**Keywords:** Software Maintenance, Quantitative Changes, Classification, Machine Learning.

**Abstract:** Software maintenance is an important stage of software development, contributing to the quality of the software. Previous studies have shown that maintenance activities spend more than 40% of the development effort, consuming most part of the software budget. Understanding how these activities are performed can support managers to previously plan and allocate resources. Despite previous studies, there is still a lack of accurate models to classify software commits into maintenance activities. In this work, we deepen our previous work, in which we proposed improvements in one of the state-of-art techniques to classify software commits. First, we include three additional features that concern the size of the commit, from the state-of-art technique. Second, we propose the use of the XGBoost, one of the most advanced implementations of boosting tree algorithms, and tends to outperform other machine learning models. Additionally, we present a deep analysis of our model to understand their decisions. Our findings show that our model outperforms the state-of-art technique achieving more than 77% of accuracy and more than 64% in the Kappa metric.

## 1 INTRODUCTION

The present article is an extension of a work presented in (Mariano et al., 2019). The development of a software project is marked by several stages. Thus, the software maintenance is important to keep certain levels of software quality (Gupta, 2017). Furthermore, previous studies (Lientz et al., 1978; Schach et al., 2003; Levin and Yehudai, 2016; Gupta, 2017; Levin and Yehudai, 2017b) have already discussed that software maintenance is the most costly step in software projects. Studies with different approaches have been present (Swanson, 1976; Mockus and Votta, 2000), seeking to understand the maintenance activities. This information can be helpful to professionals in the area to plan and allocate their resources, reduce efforts and cost, and execute projects more efficiently to improve maintenance tasks, allowing greater cost-benefit in the development of the project. In Software Engineering, we usually use a version control system (VCS) to manage changes and evaluations in the project. To understand this process, it is necessary

to classify these maintenance activities based on the history of commits. In the software industry and scientific community, three classification categories are widely used, proposed by Mockus and Votta (Mockus and Votta, 2000):

- **Adaptive:** new features are added to the system.
- **Corrective:** both functional and non-functional issues are fixed.
- **Perfective:** system and its design are improved.

Previous works that investigated commit classification into maintenance activities use only commit texts (through text analysis, such as word frequency) and reported an average accuracy of below 60% when evaluated within a unique project. Furthermore, it performed an average accuracy below 53% when the model was evaluated within several projects (Amor et al., 2006; Hindle et al., 2009). The state-of-art technique to classify commits was presented by Levin and Yehudai (Levin and Yehudai, 2017b). The authors proposed the use of Gradient Boosting Machine (GBM) (Friedman, 2001; Caruana and Niculescu-Mizil, 2006) and Random Forest (Breiman, 2001; Hindle et al., 2009) learning algorithms and take into account the commit text and source code changes (e.g., *statement added, method removed*) as the

<sup>a</sup>  <https://orcid.org/0000-0003-0351-1159>

<sup>b</sup>  <https://orcid.org/0000-0002-7571-6578>

<sup>c</sup>  <https://orcid.org/0000-0002-1523-1616>

models' features. The GBM shows an average accuracy of 72% and a Kappa coefficient of 57%. The Random Forest presents an average accuracy of 76% and a Kappa coefficient of 63%. Despite many efforts towards finding the best approach of the commit classification technique, highly accurate models are still lacking. Also, many possible features regarding commits performed by developers are not taken into account in current models. In this research paper, we studied the following problem statement:

*Problem Statement:* Current models to classify commits into maintenance activities do not use all available features regarding commits, which could potentially improve the accuracy of the models.

In this article, we investigate how a state-of-art technique classifies commits into maintenance activities. We improve the results of the state-of-art technique by calculating quantitative changes in the source code. In particular, our goal is to achieve a model with high accuracy and Cohen's Kappa coefficient. The last metric is relevant in cases when the classification categories are not balanced, i.e., when there are many more occurrences of one class in comparison to others. This scenario could mislead the results due to high accuracy; however, these results are due to imbalanced labels. We base our work on the study of Levin and Yehudai (Levin and Yehudai, 2017b), which is the current state-of-art method to classify commits into maintenance activities.

Aiming at improving both accuracy and Kappa metrics of current classification models, we propose three modifications to classify commits: (i) include the following information regarding quantitative changes in source code as additional features: total lines of code added total lines of code deleted, per commit, and the number of files changed, per commit; and (ii) use XGBoost as one of the learning algorithms. We then analyze the accuracy and kappa metrics of XGBoost and Random Forest, in comparison to the GBM and Random Forest, respectively, used in the previous paper (Levin and Yehudai, 2017b). We use XGBoost in our work due to its recent benefits and advantages over other implementations of boosting learning algorithms (Chen and Guestrin, 2015; Chen and Guestrin, 2016). Hopefully, including the three additional features and using XGBoost can increase the model's accuracy and Kappa coefficient. Third (iii), we propose the application of a more accurate analysis of the dataset and used an algorithm to select the best features of the model, and evaluate

the algorithm XGBoost (proposed on ii) in this new dataset.

The main contribution of this work is the improvement of the classification model using new software features. We believe our results can assist the development of a tool to classify commits on VCS platforms, such as GitHub. Thus, the software tool could help managers and stakeholders to track defective commits.

The remainder of this article is organized as follows. In Section 2, we present the related works. Section 3 exhibits the proposed approach. Section 4 shows the experimental setup and results. Section 5 discuss the possible applications of our results. Finally, Section 6 concludes our work and presents directions for future work.

## 2 RELATED WORK

### 2.1 Commit Classification

Classifying software maintenance commits is an always relevant challenge in the literature. The relevance given to maintenance activities is due to the large consumption of resources in this stage of software development. Swanson (Swanson, 1976) names maintenance as the "iceberg" of software development. His studies suggest that maintenance activity can consume up to 40% of the effort spent to develop the software, and this value tends to grow. These spent efforts can prevent organizations from developing new products since the main objective is to keep the software running. So understanding and classifying maintenance activities can help prevent future problems, make maintenance efficient, and cut costs.

Taking into account the importance of classification, many previous studies, with different approaches, have attempted to propose accurate models to classify commits into maintenance activities. Most works are based on the commit messages, using text analysis, such as word frequency approaches, to find specific keywords (Mockus and Votta, 2000; Fischer et al., 2003; Sliwerski et al., 2005; Hindle et al., 2009; Levin and Yehudai, 2016). For instance, Mockus and Votta (Mockus and Votta, 2000) reported an average accuracy of approximately 60% within the scope of a single project.

Levin and Yehudai (Levin and Yehudai, 2017b) combined keywords from commit comments and source code changes to classify commits. The authors used Gradient Boosting Machine (GBM) and Random Forest as underlying learning algorithms. Their

results show that both algorithms present higher accuracies than previous studies, with Random Forest performing better than GBM. However, both algorithms present regular Kappa coefficient values.

As we can note, previous studies that attempt to classify commits into maintenance activities have not taken into account many properties of the commits, such as information about the size of the commit. Previous studies (Herraiz et al., 2006; Hattori and Lanza, 2008) show some properties of the commit concerning its size, and that it can be used to help classification. In this work, we improve the model's accuracy and Cohen's Kappa coefficient (Cohen, 1960) in relation to the previous study that achieved the best results so far (Levin and Yehudai, 2017b).

## 2.2 Model Explainability

Machine learning classification techniques and models are often "black boxes", where we are unable to understand the reasons behind predictions. Previous studies have already emphasized the importance of understanding the nature of these predictions and making our models more reliable. A series of different methods have been proposed to solve this issue (Lipovetsky and Conklin, 2001; Strumbelj and Kononenko, 2013; Bach et al., 2015; Ribeiro et al., 2016; Datta et al., 2016; Shrikumar et al., 2017). The SHAP values propose in (Lundberg and Lee, 2017) unified proposed approaches, providing a single method more effective than existing ones.

In many applications understanding the prediction provided by a model can be as significant as the accuracy. More than provide predictions, usually it's important to know why the model makes some decisions and what factors influence that. In software development, have this information allow the developers to know the action that brings more efficiency to maintenance activities. For researchers, this information provides better choices of features and more transparency about how to improve the model. Thus, we also conducted a study to understand our model and the features that have the most impact.

## 3 PROPOSED APPROACH

We propose an empirical study consisting of three phases to achieve the goal. In the first phase we perform a **Literature Review** to find related works on classification of commits into maintenance activities. From previous studies (Mockus and Votta, 2000; Fischer et al., 2003; Sliwerski et al., 2005; Hindle et al.,

2009; Levin and Yehudai, 2016) we select the current state-of-the-art (SOTA) model (Levin and Yehudai, 2017b). Here, in this work, we propose a new model to overcome accuracy and kappa coefficient provided by the SOTA work by including three additional features and an implementation of XGBoost learning algorithm.

In the second phase we **Collect the Additional Features** from GitHub<sup>1</sup> via http requests to the GitHub GraphQL API<sup>2</sup>. These features bring more information on commits performed by developers and may help in the classification of commits into maintenance tasks. In Section 3.1, we explain how the labeled dataset was obtained from a previous study, in Section 3.2 we detail how the new features may be useful in distinguishing the commit categories, and in Section 3.3 we present a deeper analysis of the data.

In the third and last phase of **Experiments and Analysis** we replicate the results obtained by the SOTA using the original dataset and the same split and evaluation methods. In addition, we analyze the impact of our new features and follow the experiments with them. We split the dataset into training and test. Then, for each algorithm, the cross-validation method was used to tune the hyperparameters (Claesen and De Moor, 2015). We evaluate the training dataset using 10-fold cross-validation (Kohavi, 1995), and the average accuracy metric was reported.

Lastly, we focus on understanding the most relevant feature groups, evaluate each group separately and their combination using the XGBoost algorithm. Particularly, we evaluate the impact of the main features separately.

### 3.1 Labeled Commit Dataset

In this work, the dataset is composed of repositories hosted on GitHub, which were selected in a previous study (Levin and Yehudai, 2017b). The criteria to select the repositories following: (i) Use the Java programming language; (ii) Have more than 100 stars; (iii) Have more than 60 forks; (iv) Have their code updated since 2016-01-01; (v) Created before 2015-01-01; (vi) Had size over 2MB.

By following the mentioned criteria, 11 repositories remained in the final dataset, representing a wide domain of software projects, such as IDEs, distributed database and storage platforms, and integration frameworks. The repositories are: RxJava, IntelliJ Community, HBase, Drools, Kotlin, Hadoop, Elasticsearch, Restlet, OrientDB, Camel e Spring

<sup>1</sup><https://github.com/>

<sup>2</sup><https://developer.github.com/v4/>

FrameWork. A better description of these repositories can be seen on (Levin and Yehudai, 2017b).

Levin and Yehudai (Levin and Yehudai, 2017b) download and analyze each repository with its commit history. For each of two subsequent commits  $c_1$  and  $c_2$  ( $c_2$  has been done right after  $c_1$ ), the source code changes between them were identified and registered. These changes were first proposed by (Fluri and Gall, 2006) and they add up to 48 different types of changes.

Furthermore, for each commit, the authors inspected its comment and searched for a specific set of 20 keywords (as detailed in (Levin and Yehudai, 2017b)). The keywords are also part of the features of the model. They are represented by a binary array (of size 20) where each coordinate corresponds to a keyword and the value “1” indicates the occurrence of that keyword, while “0” indicates the absence.

Now, there are 68 features (48 + 20), corresponding to source code changes and keyword occurrences, respectively. The labeling process was manually performed by the authors from the previous study (Levin and Yehudai, 2017b) in which we based this work. Approximately 100 commits were randomly sampled from the 11 repositories and the authors classified them according to one of the three maintenance categories (corrective, adaptive, and perfective). When a commit did not present sufficient information to allow classification, the authors selected another one, until finding one which was possible to safely classify.

The authors made efforts to prevent class starvation (i.e., not having enough instances of a certain class). In case they detected a considerable imbalance in some projects classification categories they 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 1,151 manually classified commits and was made open access by Levin and Yehudai (Levin and Yehudai, 2017b), (Levin and Yehudai, 2017a).

### 3.2 Additional Features

In this work, we propose three additional features for the 11 selected repositories to be incorporated in the labeled commit dataset. As mentioned before, the features were collected through HTTP requests to the GitHub GraphQL API and refer to the changes on the files, and lines of code (LOC) changes in the source code, i.e., quantitative source code changes.

We propose the inclusion of the total files changed by the commit, total LOC added by the commit, and

total LOC deleted by the commit, giving a total of 71 features for our model. Given the 3 categories of maintenance activities, we hypothesize that the three features may reflect each class. Therefore, including these features may help in the separation of the classes, as explained below.

**Adaptive:** This maintenance activity class refers to adding new functionalities to the system, so it is more likely that added LOC is considerably higher than deleted LOC. Adding new functionalities can also cause more files to receive changes.

**Corrective:** This maintenance activity class refers to fault fixing, whence it is more likely that added LOC is very close to deleted LOC.

**Perfective:** This class refers to system design improvements. In this case, there is no previous knowledge regarding the relationship between added LOC and deleted LOC. Therefore, we may have different combinations of the features for this class.

The additional features may be extremely useful for separating *adaptive* class from the others, however, they may not be conclusive regarding the other classes. As we can see in Figures 1, 2 and 3, in fact, added LOC is considerably higher in *adaptive* class, changed files is less significant in *corrective*, while deleted LOC does not present a strong difference.

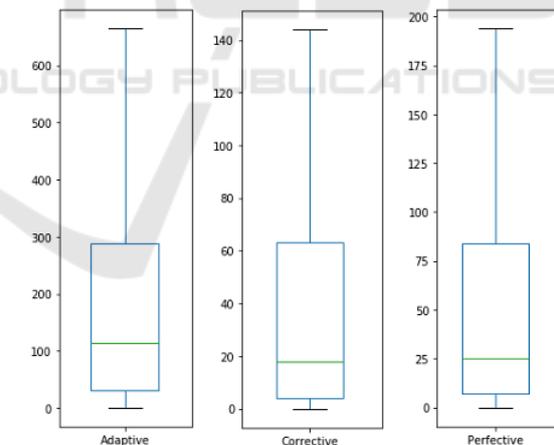


Figure 1: Added lines of code.

These boxplots were generated based on the values of the features collected for the repositories that constitute our dataset. We decided to include all three features since their co-occurrences may be helpful to separate the maintenance classes.

### 3.3 Dataset Overview

To better understand the behavior and characteristics of the dataset, we present a deeper analysis of the

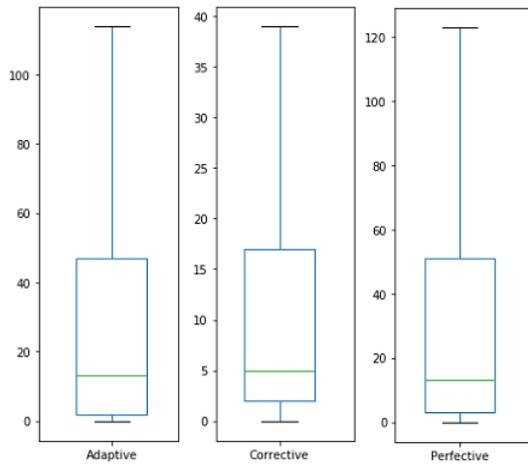


Figure 2: Deleted lines of code.

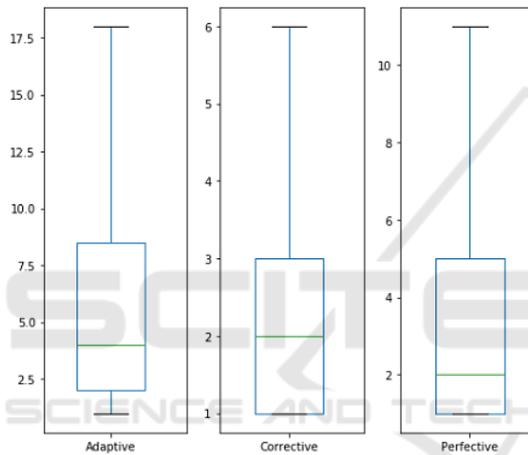


Figure 3: Changed Files.

data. Here we make the analysis and create two new datasets to be used in our experiments. First, we check all the commits, one by one, search for any problem, and find two appearances of duplicate commits.

1. Rows 503 and 523 are identical with all their columns, including the *commitID* and *label*, so we decided to delete line 523 and keep line 503.
2. Rows 96 and 97 also have duplication in all columns, including the *commitID* however the *label* for line 96 was provided as corrective and line 97 as perfective, we decided to exclude both lines since the correct classification of this commit is not accurate.

Next step, to improve the analysis, we apply the Pandas Profiling<sup>1</sup>, a Python tool that generates a report about a dataset. They show the column-by-column information interactively, allowing a more detailed and

<sup>1</sup><https://pypi.org/project/pandas-profiling/>

quantitative visualization of each feature in addition to the existing correlation between them. Among the main information, besides size and data type, we receive information like missing values, most frequent values, descriptive and quantile statistics, data distribution, and variance. Thus, we could identify meaningful information about the data.

The feature “*PARENT\_INTERFACE\_CHANGE*” presents in all rows a constant value equal to 0. In this way, the feature was automatically rejected by the tool. It is important to state that the feature is only rejected concerning this dataset since none of the instances present this information, which makes it useless in model production. However, for a more robust dataset, with new instances that present the characteristics of this feature, it could return to the model.

We observe that 80% (56) of the features have more than 86% (1000) of that data are constant, more specifically equal to zero. This occurred because the keywords are binary features until each instance has only a few source code changes, but this can have many influences in the model. For example, The feature “*ADDING\_CLASS\_DERIVABILITY*”, have only one instance with a value different than zero. Although, in our analysis, we have not noticed a high correlation with the features. After all the removals, rows, and columns, now we have a second dataset, that we call “Complete”, with 1,148 lines and 70 features(47 source code changes + 20 keywords + 3 additional).

Searching the features that were more important to a model, we also apply a Supervised Tree-based Feature Selection, and select the more impactful features for the Classification. We chose Random Forest for this selection because it is the best performing algorithm in the previous work and is different from the XGBoost algorithm used for analyzing. Selecting the more impactful features can help find the true importance of these features in the classification. After applying this supervised feature Selection, we have a third dataset, that we call “Select” with 1,148 lines and 21 features. Of these 21, we have 11 source code changes, 7 keywords, and the 3 additional features. It is important to note that the selection model itself selected the additional features, which helps to show that the addition of features with the quantitative change can bring a significant result. The composition of Both Complete and Select dataset is classified as 43.5% (499 instances) were corrective, 35.1% (403 instances) were perfective, and 21.4% (246 instances).

That will be important to give a brief explanation of each feature on the Select dataset to understand our experiments. First, we have the 3 new features that we

propose, “additions”, “deletions” and “changedFiles” that represent the number of lines added, number of lines delete and a number of files change, respectively, we can see more explanation on section 3.3. Second we have the 7 keywords as described in Section 3.2 they indicate if the word appeared in the message, they are the following words: “add”, “allow”, “fix”, “implement”, “remov”, “support” and “test”. Finally, we have the 11 features of source code changes, a more detailed explanation can be found on (Fluri and Gall, 2006), follow:

- `ADDITIONAL_FUNCTIONALITY`: add functionality on code.
- `REMOVED_FUNCTIONALITY`: remove functionality on code.
- `ADDITIONAL_OBJECT_STATE`: add of attributes that describe the state of an object.
- `ALTERNATIVE_PART_INSERT`: the inserting or deleting that does not have any impact on the overall nested depth of a method.
- `COMMENT_INSERT`: inserting comments.
- `CONDITION_EXPRESSION_CHANGE`: update operation of a structured statement.
- `DOC_UPDATE`: update on a document.
- `STATEMENT_DELETE`: delete operation in the entities control structure, loop structure, and statement.
- `STATEMENT_INSERT`: insert operation.
- `STATEMENT_UPDATE`: update operation.
- `STATEMENT_PARENT_CHANGE`: move operation.

## 4 EXPERIMENTS AND RESULTS

### 4.1 Replicating Results

Before proceeding with the experiments, it is important to check if the results acquired by the SOTA are replicable. For this reason, we split the original dataset into a training dataset and a test dataset. The split is the same made by the SOTA, 85% on training, and 15% on the test. We performed the split by using scikit-learn library (Pedregosa et al., 2011) from Python language, which use the label to balance the class distributions. The detail of the split label is in Table 1, and the split by commits per project is on Table 2.

Since we propose a new algorithm, we train the model with the training dataset using XGBoost. Then,

Table 1: Split labels.

	Corrective	Perfective	Adaptive
Train	424	345	209
Test	76	59	38

Table 2: Split commits.

Project	Train	Test
hadoop	100	12
drools	97	17
intellij-community	93	16
ReactiveX-RxJava	88	12
spring-framework	87	13
kotlin	86	15
hbase	86	23
elasticsearch	86	14
restlet-framework-java	86	15
orientdb	85	17
camel	84	19

the trained models were evaluated using the test dataset. The test dataset did not take part in the model training process. We do the training using the same models, as proposed by the SOTA. The best performing compound model for each classification algorithm was evaluated on the test dataset.

Our results are very close to the results reported by Levin and Yehudai (Levin and Yehudai, 2017b). We obtained 76% of accuracy and 63% of Kappa from Random Forest, and 73% of accuracy and 58% of Kappa from XGBoost. We conclude that the SOTA is replicable and XGBoost is an option for evaluation.

### 4.2 Impact of Features

Now we take our second dataset (complete) and analyze the impact that our additional features have on the model. For better visualization, we did an impact test of each feature separately, and their combinations (Changed Files: CF, added LOC: AL, and deleted LOC: DL). The XGBoost is the default algorithm from the Shap value. We use the Shap to show the importance of the features in section 4.7. So, we decide to use also XGBoost algorithm for visualization of the impact of the features. For this test, we use a simple 10-fold cross-validation across all datasets without using hyperparameters. The techniques were made using the scikit learn library (Pedregosa et al., 2011). We can see the difference in Table 3.

### 4.3 XGBoost

We run XGBoost on the training dataset and evaluate it by using 10-fold cross-validation. A grid-search of hyperparameters was passed to the evaluation method in order to find the best hyperparameters.

Table 3: Features Impact.

	Accuracy	Kappa
NaN	71,69	55,89
CF	71,60	55,74
AL	72,30	56,99
DL	71,62	55,57
CF, AL	72,50	57,26
CF, DL	71,36	55,23
AL, DL	72,66	52,55
<b>CF, DL, AL</b>	<b>72,31</b>	<b>56,81</b>

The main parameter to tune is the maximum depth of a tree in the XGBoost. Table 4 shows a simple example of some variations in the hyperparameters *colsample\_bytree*, or simply *colsample* (subsample ratio of columns), and *max\_depth* (maximum tree depth), with 150 iterations. The best parameters are in bolded text. We also verified the model performance during the training by plotting some graphs for several hyperparameters. We choose accuracy to decide which parameters we should select.

Table 4: Example of hyperparameters grid for XGBoost.

max_depth	colsample	Accuracy	Kappa
1	0.4	0.6952	0.5258
1	0.7	0.7004	0.5334
<b>3</b>	<b>0.4</b>	<b>0.7443</b>	<b>0.6011</b>
3	0.7	0.7280	0.5766
6	0.4	0.7239	0.5683
6	0.7	0.7259	0.5721

As we can observe in the presented table 4, the best hyperparameters were 0.4 (*colsample\_bytree*) and 3 (for *max\_depth*). Using these parameters, the model presented 74.43% of accuracy and 60.22% of Kappa in training. With these values in hand, the model was evaluated in the test dataset. It is important to remind that this dataset was not taken into account during the training phase so that we can have a more precise evaluation of the model. In Table 5, we can see the predicted classes by the model in the confusion matrix. This matrix presents the corrected and wrong predictions, from which we are able to calculate the accuracy and Kappa.

Table 5: XGBoost confusion matrix for test dataset.

Prediction	True class		
	Adaptive	Corrective	Perfective
Adaptive	26	1	7
Corrective	5	63	13
Perfective	6	11	40

By inspecting the confusion matrix for XGBoost, our model presented an *Accuracy* of 75.7% and a *Kappa*

*Coefficient* of 60.7% regarding the commit classification into adaptive, corrective, and perfective maintenance classes.

#### 4.4 Random Forest

Similarly to XGBoost, we run Random Forest in the training dataset and evaluated it by using 10-fold cross-validation. We also use a grid-search to find the best hyperparameters. In particular, the main parameter to tune is the number of predictors that are randomly selected for each tree. This technique is used to make sure that trees are uncorrelated. The hyperparameter tuning is also done by the scikit-learn library (Pedregosa et al., 2011). Table 6 presents a simple example of some variations in the hyperparameter *mtry* (number of randomly selected predictors). The best parameter is in bolded text in Table 6. As for XGBoost, we also verified the model performance during the training by plotting graphs for different values of the *mtry*. We choose accuracy to decide which parameters we should select.

Table 6: Example of hyperparameters grid for Random Forest.

mtry	Accuracy	Kappa
2	0.6546	0.4384
<b>35</b>	<b>0.7013</b>	<b>0.5355</b>
69	0.6868	0.5123

Table 6 show that the best hyperparameter value was 35 (*mtry*). This means that for each tree in the forest, 35 features are randomly selected among the 70. Using this parameter, the model presented 70.13% of accuracy and 53.55% of Kappa in training. With this parameter value in hands, the model was evaluated in the test dataset. In Table 7, we can see the predicted classes by the model in the confusion matrix.

Table 7: Random Forest confusion matrix for test dataset.

Prediction	True Class		
	Adaptive	Corrective	Perfective
Adaptive	25	2	6
Corrective	3	61	7
Perfective	9	12	47

From the observe of confusion matrix for Random Forest, our model presented an *Accuracy* of 77.32% and a *Kappa Coefficient* of 64.61% regarding the commit classification into adaptive, corrective, and perfective maintenance classes.

## 4.5 Summary of Results

The models proposed in this work achieved better results in comparison to the SOTA technique to classify commits into maintenance activities. This is indicative that, in fact, there are ways to improve this task. Table 8 reveals that, for each learning algorithm, how the accuracies and Kappa coefficients are achieved by our models compare to the previous study.

Table 8: Comparison of SOTA and proposed improvements.

	SOTA		OURS	
	GBM	RF	XGBoost	RF
Accuracy	0.7200	0.7600	0.7570	0.7732
Kappa	0.5700	0.6300	0.6070	0.6461

We note that XGBoost presented an increase in both accuracy and Kappa coefficient in comparison to the Gradient Boosting Machine used in the previous work. While Levin and Yehudai (Levin and Yehudai, 2017b) achieved 72.0% of accuracy and 57.0% of Kappa, our model achieved 75.7% of accuracy and almost 61.0% of Kappa. These results show that including added lines of code and deleted lines of code as features may be useful for classifying commits. In addition, using advanced implementations of boosting algorithms, such as XGBoost, in fact, can improve model performance.

Regarding Random Forest, we can observe that the difference from our results to the past one was smaller, but still, we achieved higher accuracy and Kappa Coefficient. While the previous model presented 76.0% of accuracy and 63.0% of Kappa, our model reached an accuracy of 77.3% and Kappa of 64.6%.

In summary, our model presented higher accuracies and Kappa coefficients for both algorithms. For XGBoost, our model achieved an increase of almost 4% for accuracy and Kappa. Regarding the Random Forest, we increased the accuracy by 1.3% and the Kappa by almost 2%.

## 4.6 Dataset Measures

It is possible to observe that depending on the way we evaluate the model, using keywords or source code changes, our results may be different. Now we look at the real impact using different datasets. First, we evaluate the types of features separately, and then combine them, and finally compare with our two new datasets, “Complete” and “Select”. We can see all comparisons in Table 9.

Table 9: Dataset Comparison.

	Accuracy	Kappa
Keywords	72.72	57.82
Changes	50.87	21.99
Quantitative	49.02	18.56
Keywords + Changes	73.64	59.16
Keywords + Quantitative	73.35	58.75
Changes + Quantitative	56.47	31.01
<b>Combine</b>	<b>75.72</b>	<b>62.41</b>
Complete	74.45	60.11
Select	71.16	54.78

## 4.7 Model Understandability

It is important to analyze which group of features are most important for the model. In Table 9, we observe these features importance. The “Quantitative” group of features proposed in this paper proves to be slightly efficient in the prediction. Only a group of 3 features achieved accuracy and kappa very close of the 48 features of Source code changes (“Changes”). Moreover, we can see that “Keywords” are essential for high accuracy. The “Select” dataset presented in this paper has only 22 features, which is less than a third of the original “Combine” dataset and still achieved relevant results, and close to the dataset with all features.

It is possible to notice that some features are more important than others in the model. Understanding only the groups with the greatest impact is not enough to understand the whole model. Comprehend the impact of each feature, separately, can be done with a model understandability.

To explain our model decisions, we use a technique development, proposed and present in (Lundberg and Lee, 2017), which uses SHAP values. This technique is a unified measure of feature importance, to unveil the “black boxes” in the classification models. The SHAP values work with any tree-based model, they calculate the average of contributions across all permutation of the features. In summary, they can show how each feature contributes, positively or negatively to a model. We use the “Select” dataset for this analysis.

Figure 4 presents a SHAP summary plot, showing the impact of each feature on the model. We observe that 7 features are important for the model, ranked in descending order, the most important feature “support” is the first in the plot, and “ADDITIONAL\_FUNCTIONALITY”(ADD\_FUNCT) is the second. On the horizontal, we have the module of the impact on the prediction model. The closer the values of 0, the smaller its impact. In contrast, the farther from 0 means a bigger impact. Therefore, analyzing Figure 4, the “support” feature has a bigger impact, with an average of 0.08 of magnitude. In this

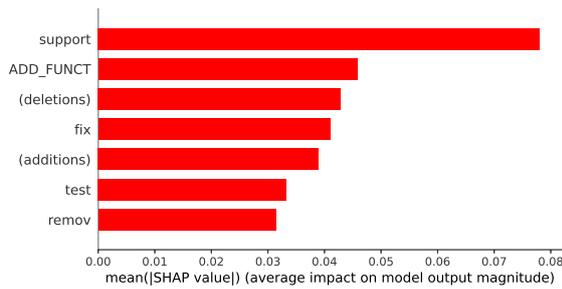


Figure 4: The most important features for the model.

way, we can conclude that has the “support” word on the commit message follows a significant impact on the classification of the model. Besides that, corroborating with Table 9, we note that keywords are the most important group on the model, representing 4 of the 7 best features. Following this rationale, we have 2 of our proposed features and 1 of the source code changes between the most important features for the prediction.

## 5 APPLICATIONS

Levin and Yehudai investigated the amount of a commit that a given developer made in each of the maintenance activities and suggested the notion of a developer’s maintenance profile (Levin and Yehudai, 2016). Their model to predict developer maintenance profile could be supported by our proposed model to classify commits into maintenance tasks, possibly with yield higher accuracy in their prediction.

Another application of our proposed model is in the identification of anomalies in the development process. It is important to manage the maintenance activities performed by developers, i.e., the volume of commits made in each maintenance category. Monitoring unexpected behavior in maintenance tasks would assist managers to plan and allocate resources in advance. For instance, lower adaptive activity may indicate that the project is not evolving as it was expected, and lower corrective activity may suggest that developers are neglecting fault fixing. Identifying the root causes of problems may aid the manager to overcome them. Furthermore, recognizing maintenance patterns in successful projects may be useful as guidelines for other projects.

Building a software team is a non-trivial task given the diverse factors involved with it, such as technological and human aspects (Gorla and Lam, 2004; dos Santos and Figueiredo, 2020). Commit classification may help to build a more reliable and balanced developer team regarding the devel-

oper maintenance activity profile (Levin and Yehudai, 2016; dos Santos and Figueiredo, 2020). This can be done create an API or software linked to the version control to identify and classify the commit automatically. When a team is composed of more developers with a specific profile (e.g., adaptive) than others, the development process may be affected, and the ability of the team to meet usual requirements (e.g., developing new features, adhering to quality standards) could also be negatively impacted.

## 6 CONCLUSIONS

Software maintenance is an extremely relevant task for software projects, and it is essential for the whole software life cycle and operation. Maintenance is shown to consume most of the project budget. Therefore, understanding how maintenance tasks are performed is useful for practitioners and managers so that they can plan and allocate resources in advance.

In the present article, we proposed improvements on a SOTA approach used to classify commits into software maintenance activities. In particular, we proposed the adoption of three new features that measure quantitative changes of source code, since the SOTA do not use that type of features on their commit classification. We also proposed the use of the XGBoost algorithm to perform commit classifications. Also, we carried out experiments using an already labeled commit dataset to evaluate the impact of our proposed improvements on commit classification.

Experimental results showed that our proposed approach achieved 76% accuracy and 61% of Kappa for the XGBoost, an increase of 4% in comparison to the past study. Also, our Random Forest achieved 77.3% and 64.6% of accuracy and Kappa, respectively, with an increase of 1.3% and approximately 2% related by SOTA. We notice that quantitative changes can be helpful to understand the characteristics of a commit. Furthermore, the use of additional features have become extremely relevant in our machine learning model. Thus, in terms of model understandability, we could determine that the additional features are on the top of the most important features in the classification model.

As future work, we intend to evaluate other features related to software commits and use other datasets with a larger number of commits to improving the accuracy of the classifiers. We also intend to evaluate other classification algorithms using different evaluation metrics, e.g., precision and recall, to improve our understanding of the behavior of the classifier. Commits have more metadata that was not

analyzed in this article due to the scope of the application and the intended comparison with the existing approach. Moreover, we intend to use NLP (Natural Language Processing) to investigate the commit text based on maintenance activities. Thus, we could generate a classification approachable to classify the commits automatically based on the labels discussed in this article. Finally, we intend to use other categories (i.e., activities) proposed in the literature to generalize the results.

## ACKNOWLEDGEMENTS

The present work was carried out with the support of the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brazil (CAPES) - Financing Code 001. The authors thank the partial support of the CNPq (Brazilian National Council for Scientific and Technological Development), FAPEMIG (Foundation for Research and Scientific and Technological Development of Minas Gerais), and PUC Minas.

## REFERENCES

- Amor, J., Robles, G., Gonzalez-Barahona, J., Gsyc, A., Carlos, J., and Madrid, S. (2006). Discriminating development activities in versioning systems: A case study. In *Proceedings of the 2nd International Conference on Predictor Models in Software Engineering*, PROMISE'06.
- Bach, S., Binder, A., Montavon, G., Klauschen, F., Müller, K., and Samek, W. (2015). On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PLoS ONE*, 10.
- Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.
- Caruana, R. and Niculescu-Mizil, A. (2006). An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML'06, pages 161–168.
- Chen, T. and Guestrin, C. (2015). XGBoost: Reliable large-scale tree boosting system. In *Proceedings of the NIPS 2015 Workshop on Machine Learning Systems*, LearningSys'15.
- Chen, T. and Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd International Conference on Knowledge Discovery and Data Mining*, KDD'16, pages 785–794.
- Claesen, M. and De Moor, B. (2015). Hyperparameter search in machine learning. In *Proceedings of the 11th Metaheuristics International Conference*, MIC'15.
- Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46.
- Datta, A., Sen, S., and Zick, Y. (2016). Algorithmic transparency via quantitative input influence: Theory and experiments with learning systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, SP'16, pages 598–617.
- dos Santos, G. E. and Figueiredo, E. (2020). Commit classification using natural language processing: Experiments over labeled datasets. In *Proceedings of the 23rd Iberoamerican Conference on Software Engineering*, CibSE'20.
- Fischer, M., Pinzger, M., and Gall, H. (2003). Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, ICSM'03, pages 23–32.
- Fluri, B. and Gall, H. C. (2006). Classifying change types for qualifying change couplings. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC'06, pages 35–45.
- Friedman, J. (2001). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29(5):1189–1232.
- Gorla, N. and Lam, Y. W. (2004). Who should work with whom?: Building effective software project teams. *Communications of the ACM*, 47(6):79–82.
- Gupta, M. (2017). Improving software maintenance using process mining and predictive analytics. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, ICSME'17, pages 681–686.
- Hattori, L. P. and Lanza, M. (2008). On the nature of commits. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE'18, pages 63–71.
- Herraiz, I., Robles, G., Gonzalez-Barahona, J. M., Capiluppi, A., and Ramil, J. F. (2006). Comparison between slocs and number of files as size metrics for software evolution analysis. In *Proceedings of the Conference on Software Maintenance and Reengineering*, CSMR'06, pages 8 pp.–213.
- Hindle, A., German, D., Godfrey, M., and Holt, R. (2009). Automatic classification of large changes into maintenance categories. In *Proceedings of the 17th IEEE International Conference on Program Comprehension*, ICPC'09, pages 30–39.
- Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, IJCAI'95, page 1137–1143.
- Levin, S. and Yehudai, A. (2016). Using temporal and semantic developer-level information to predict maintenance activity profiles. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, ICSME'16, pages 463–467.
- Levin, S. and Yehudai, A. (2017a). 1151 commits with software maintenance activity labels (corrective, perfective, adaptive). Available: <https://doi.org/10.5281/zenodo.835534>.
- Levin, S. and Yehudai, A. (2017b). Boosting automatic commit classification into maintenance activities by

- utilizing source code changes. In *Proceedings of the 13rd International Conference on Predictor Models in Software Engineering*, PROMISE'17, pages 97–106.
- Lientz, B. P., Swanson, E. B., and Tompkins, G. E. (1978). Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471.
- Lipovetsky, S. and Conklin, M. (2001). Analysis of regression in game theory approach. *Applied Stochastic Models in Business and Industry*, 17:319 – 330.
- Lundberg, S. M. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 4768–4777.
- Mariano, R. V. R., dos Santos, G. E., V. de Almeida, M., and Brandão, W. C. (2019). Feature changes in source code for commit classification into maintenance activities. In *Proceedings of the 18th IEEE International Conference on Machine Learning and Applications*, ICMLA'19, pages 515–518.
- Mockus, A. and Votta, L. G. (2000). Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance*, ICSM'00, pages 120–130.
- Pregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.
- Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "Why Should I Trust You?": Explaining the predictions of any classifier. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics*, pages 97–101.
- Schach, S., Jin, B., Yu, L., Heller, G., and Offutt, J. (2003). Determining the distribution of maintenance categories: Survey versus measurement. *Empirical Software Engineering*, 8:351–365.
- Shrikumar, A., Greenside, P., and Kundaje, A. (2017). Learning important features through propagating activation differences. In *Proceedings of the 34th International Conference on Machine Learning*, ICML'17, page 3145–3153.
- Sliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? *Software Engineering Notes*, 30(4):1–5.
- Strumbelj, E. and Kononenko, I. (2013). Explaining prediction models and individual predictions with feature contributions. *Knowledge and Information Systems*, 41:647–665.
- Swanson, E. B. (1976). The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE'76, pages 492–497.