**EPJ manuscript No.**
(will be inserted by the editor)

# Commit Message Classification for Betterment of Software Development

Mohamed Wiem Mkaouer[1] and Priyadarshni Sagar[2a]

[1] Assistant Professor of Software Engineering
Department of Software Engineering
*mwmvse@g.rit.edu*
[2] MD Data Science
Department of Software Engineering
*ps1862@rit.edu*

Received: date / Revised version: date

**Abstract.** The software industry is constantly evolving, and requirements are changing every day.Monitoring the software development and maintenance of software is a deciding factor for resource allocation, time requirements, a milestone to be followed in order to complete the project . As a part of software maintenance often developers need to add new features to the existing software and perform refactoring. To modify the quality of software without altering its external behaviour is quite difficult thus the software development industry is seeking for an automated way to predict the necessary refactoring. In the open source community a developer's impression of using a fairly limited vocabulary and restricted structure while writing the commit messages creates a problem for software maintenance, this impacts the ability of developers to modify existing software. In this paper, we are proposing a method to predict refactoring type and detect usual, unusual commits based on refactoring type, time when the commit was made, location, author, class,etc. We have defined a set of heuristics, used different machine learning and deep learning techniques to predict the refactoring type with commit messages and code metrics.

## 1 Introduction

Having a better understanding of maintenance activities performed in a source code of software could help project leads to reduce uncertainty and improve cost-effectiveness by planning and preallocating resources towards software maintenance.These maintenance activities are classified as corrective, perfective and adaptive [1]. Previous research done in the maintenance improvement classified the commits into three categories. Although this classifier gained the advancement in the maintenance activities, it is failed to fulfill all the requirements of software maintenance team like it is not revealing which commits are causing software improvements, which commits are for documentation, which commits are fixing the bugs, which commits are affecting the system and dependencies, refactoring, style-related commits. The method proposed in the system is introducing the models to classify commits and predict refactoring types.
Software refactoring is used to improve software quality by applying changes on internal structure of software without altering its external behavior. Often developers need to change the existing software to add new features, edit the existing features, fix the problem with the system. While performing these changes they are uncertain about the refactoring type. The motivation behind this study is to automate the process of identifying refactoring type. Developers can identify the refactoring type by doing some research of software or by having conversation with team members using this software, by their intuition, with background knowledge of software, more often they need to rely on expertise to identify refactoring opportunities, however this process is tedious and success is not guaranteed. Thus automated ways of identifying refuting types will help developers, maintenance teams and save their time. Studies done in the past used various methods to predict refactoring type, such as deep learning, machine learning, convolutional neural network[2][3][4], few experiments have used AI based techniques to predict the refactoring type.
The approach presented in this paper used machine learning algorithms and deep learning to predict refactoring type. To begin with we considered only commit messages as input for machine learning algorithms but developers' limited

[a] *Present address:* Insert the address here if needed

vocabulary usage made it difficult for algorithms to achieve maximum accuracy. There are a number of studies that revealed the correlation between code smell metrics and refactoring type[2], thus we started our experiments with code metrics as input and followed by this we also tested a deep learning model that considered code metrics and commit messages as input.

Commits are the short summaries written by the developers describing the changes made during the development process by them. These changes to the source code of a software system have different purposes e.g. to add a new feature, fix a bug, to perfect a system. The repository has all the commits made by the developers. Commits can be grouped according to their purposes. The research [5] *[Eman: please add reference:done]* has been done in this field of commit message classification where commits are classified based on the reasons for them by considering attributes related to the size, density, and text of commits. In open source community developer's impression of using a limited vocabulary and restricted structure while writing the commit messages creates a problem for software maintenance. This problem might lead to weakening the ability of decision making for project leads and managers.

Three experiments were performed in this study to consider different sets of inputs. Our models will introduce more classes to classify the commits with reduced dimensionality of attributes and it also considers the structural and relational properties of commits for better results. In future, this system will also classify the commits as unusual or not in order to check the quality of code contained in the commit. Unusual commits question the quality of the commit. Our model will predict the developer's maintenance profiles, and this will benefit from a more precise classification by considering more classes with reduced dimensionality of attributes. Introducing more classes and consideration of structural, relational properties possibly yield higher prediction quality. Monitoring for unexpected spikes in maintenance activities and investigating the causes behind them would help managers and maintenance to plan ahead of the resource allocation. For example, higher corrective profiles imply there is a high bug count, higher documentation profiles imply that the project is well documented. Finding such causes will be a significant factor to check the project's health. We believe that by implementing a multiclass classifier it would be possible to build a reliable developer's maintenance activity profile and keep balance in the development process.

The methods proposed in this paper answered the research questions such as how accurate the commit messages are in predicting the refactoring class? How accurately the commit messages with code metrics can predict the refactoring class? How does the code metrics influence refactoring class prediction? And which refactoring classes were most accurately classified by each method? Deep learning algorithms to predict the refactoring class with commit messages and combination of code metrics, commits gave us poor accuracy whereas supervised machine learning algorithms trained with code metrics as input resulted in the most accurate classifier. Accuracy per class varied for each method and algorithm, this was expected.

Supervised machine learning model trained on the SMOTE-transformed training dataset gave 75% accuracy, and predicted inline class with 99% of accuracy. Out of all supervised algorithms, only the random forest algorithm achieved good accuracy. LSTM models trained on commit messages and combination of inputs resulted in poor accuracy. *[Eman: please add one paragraph to summarize the research questions and another paragraph to summarize the results: done]* The remainder of this paper proceeds by discussing the different commit message classification and refactoring prediction techniques implemented by other researchers where general outline will be given about research done in the same field, the approach followed to accomplish the aim of the study and how implementation has been done. The last section will mainly focus on results and future work.

## 2 Related Work

### 2.1 Background

Software development and maintenance requires the number of changes like adding a new feature,fixing the bug, improving the performance. These changes have different aims and purposes.Developers group these changes into commits. Developers, managers are often interested in analyzing these changes made to the software. If we classify these changes into meaningful groups based on the type of change will help the maintenance team to reach the conclusion of decision making. There are three main categories of maintenance activities: 1.Corrective which fixes the faults,functional, nonfunctional aspect of the software, 2. Perfective which improves the system, 3. Adaptive introduces new features into the existing system. Classifiers are built previously for commit message classification by considering these categories achieved accuracy, but the classes are not specific in this method which motivated our work. Although the classifiers gained the advancement in the maintenance activities, it is failed to fulfill all the requirements of software maintenance team like it is not revealing which commits are causing software improvements, which commits are for documentation, which commits are fixing the bugs, which commits are affecting the system and dependencies, refactoring, style related commits. Our model proposed in this paper will introduce more classes to classify the commits with reduced dimensionality of attributes and it also considers the structural and relational properties of commits for better results. LSTM based model and supervised learning models presented in this paper are automating the refactoring prediction to help developers and maintenance team while choosing the correct refactoring.

In future, This system will classify the commits as unusual or not in order to check the quality of code contained in the commit. Unusual commits question the quality of the commit.

## 2.2 Approaches for classification

*[Eman: This section has number involved between words that we need to clean.] [Eman: For some of the studies in this section, it is better to cite while mentioning the author name (e.g., FAuthor et al. [] ) :done , added in few sections]*

### 2.2.1 Deep Learning

Implementing a deep learning approach for commit message classification resulted in high accuracy. For active learning of classifier, unlabeled dataset of commit messages is created and labeling is done after performing the feature extraction using Term Frequency Inverse Document. The approach followed the steps like dataset construction which includes a text prepossessing and a feature extraction step, Multi-label active learning phase during which a classifier model is built then evaluated and unlabeled instances are queried for labeling by an oracle, and classification of new commit messages. GitProc [6] is used for data collection from 12 open-source projects. Classifier using active learning is tested by measures such as hamming loss, precision, recall and F1 score. Active learning multi-label classification technique reduced the efforts needed to assign labels to each instance in a large set of commits. The classifier presented in the study by Gharbi, Sirine, et al. [7] can be improved by considering the changes of nature of the commits using commit time and their types also auto- mated commit classification written in different languages i.e. multilingual classification is gap for Betterment. Mining the open source repositories is difficult for the software engineers because of the error rate in the labeling of commits. Prior to this work key word-based approach is used for bug fix commits classification. The method implemented by Zafar et al. [1] uses the deep learning models Bidirec- tional Encoder Representations from Transformers(BERT) which can understand the context of commits and even the semantics for better classification by creating a hand labeled dataset and semantic rules for handling complex bug fix commits which in turn reduced the error rate of labeling by 10%.The technique [1] analyzed git commits to check if they are bug fix commits are not, this will help the development team to identify the future resources and achieve project goals in time by integrating NLP and BERT for bugfix commit classification. This Implemented approach is based on fine-tuning with the deep neural network which encodes the word relationships from the commits for the Bug-Fix identification task.

### 2.2.2 Resampling Technique

Often commit message datasets are imbalanced in nature and It is difficult to build a classifier for such a dataset, it might cause undersampling and oversampling. The method proposed in [8] classifies a commit messages extracted from GitHub using the multiple resampling technique for highly imbalanced dataset resulting in improvements in classification over the other classifiers. Imbalanced dataset often causes problems with the machine learning algorithm. There are three variants of the resampling, under sampling, over sampling and hybrid sampling. undersampling method balances the class distribution to reduce the skewness of data by removing minority classes, whereas oversampling duplicates the examples from minority classes to minimize skewness and hybrid sampling uses the cobination of undersampling and oversampling. *[Eman: we need to add definition for each of these techniques: done]*. All these methods tend to maintain the goal of statistical resampling by improving the balance between the minority and majority classes. Study done in [8] fist creates the feature matrix and resampling is done by imbalanced learn sampling method, here 10-fold cross validation is used to ensure the consistent result. From the research [8] the questions concerning the development process like do developers discuss design? Is answered.

### 2.2.3 DeepLink: Issue-Commit Link Recovery

For the online version control system like GitHub links are missing between the commits and issues. Issue commit links plays an important role in software maintenance as they help to understand the logic behind the commit and make the software maintenance easy. Existing systems for issue commit link recovery extracts the features from issue report and commit log but it sometimes results in loss of semantics. Xie, Rui, et al.[9] proposed the design of software that captures the semantics of code and issue related text, further it also calculates the semantics similarity and code similarity using Support Vector Machine (SVM) classification. Deeplink followed the process in order to calculate the semantic and code similarity which includes data construction, generation of code embeddings, similarity calculation and feature extraction. Result is supported from [9] by the experiment performed on 6 projects which answered the research questions relying on the effectiveness of deeplink in order to recover the missing links, effects of code context and semantics on deeplink giving 90of F1-measure.

### 2.2.4 Code Density for Commit Message Classification

Classification of commits support the understanding and quality improvement of the software. The concept introduced in [10] uses code density i.e. ratio between net and gross size of the code change. where net size is the size of the unique code in the system and gross size is size of everything, including clones, comments, space lines, etc. Answers for the question are revealed by [10], what are the statistical properties of commit message dataset, is there any difference between cross and single project classification, does classifier performs better by considering the net size related attributes? Are the size and density related features suitable for commit message classification? And developed a git density tool for analyzing git repositories. This work can be extended by considering the structural and relational properties of commits while reducing dimensionality of Features.

### 2.2.5 Boosting Automatic Commit Classification

There are three main categories of maintenance activities, predictive, adaptive and corrective. Better understanding of these activities will help managers, development team to allocate resources in advance. Previous work done on commit message classification, mainly focused on single project. The work done by Levin et al. [5] presented a commit message classifier capable of classifying commits across different projects with high accuracy. 11 different open source projects were studied, and 11513 commits were classified with high kappa value and high accuracy. The results from [5] showed that when the analysis is based on word frequency of commits and source code changes, model boosted the performance. It considered the cross-project classification. Methods are followed by gathering the commits and code changes , sampling to label the commit dataset, developing a predictive model and training on 85% data and testing on 15% of test data from same commit dataset , [5] used naïve bayes to set the initial baseline on test data. This system of classification motivated us to consider the combinations of maintenance classes like predictive + corrective. In order to support the validation of labeling mechanisms for commit classification and to generate a training set for future studies in the field of commit message classification work presented by Mauczka, Andreas, et al [11] surveyed source code changes labeled by authors of that code. For this study seven developers from six projects applied three classification methods to evident the changes made by them with meta information. Automated classification of commits can be possible by mining the repositories from open source like git even though precision, recall can be used to measure the performance of classifier only the authors of commits knows the exact intent of change.

### 2.2.6 Classification of unusual commits

Transparent open sources such as GitHub helps the developers to get notified about the changes made to software and maintenance carried in that project, but overwhelming notifications may distract the developers and they may lose the important changes while getting attention towards unnecessary changes. To deal with this problem a novel approach is proposed by Goyal, Raman, et al [12] which built an anomaly detection model to classify the unusual commits in the repository of a project. This model is able of classifying the large commits, commits made at the unusual time, commits made on rarely used files from the project repository. The proposed model rates the commit based on the unusual score and it decides whether to notify the developer or not about the same. This model identifies the breaking changes such as cru- cial fixes to vulnerabilities, relevance based prior activity, unusual changes to repository. Statistical way to classify outliers is considered along with various commit characteristics. This helped the maintenance team and developers as unusual commits also questions the quality of code. Our work will focus on unusualness of commits from project repository to ensure the quality of software.

### 2.2.7 Tools for Classifying GitHub Commits

The research from [6] presented a tool for processing and classifying the GitHub commits based on regular expressions and source code blocks. This tool downloads projects and extract the project history including the source code information and development time bug fixes and it is capable of handling both single line and block source code written indifferent languages like C, C++, Java and Python can handle diverse code structures. Gitproc also considers the project log classification based on different keywords like bug classification, keyword-based log classification. GitcProc is a tool which extracts the information from git log metadata and source code diffs. Furthermore, one can consider improving GitProc by adding support for support constructors and destructors defined outside the class. Class and function declaration in KRC style, handle very complex scope changes in C,C++ and Java.

#### 2.2.8 Predict refactoring type

As a part of software maintenance activity, often a developer needs to add a new feature to the existing software, modifying the software quality without affecting it's external feature is a refactoring. [2]. Refactoring identification is crucial as it impacts the quality of software, developers decide the refactoring opportunity based on their knowledge, expertise, thus there is a need for an automated way to predict the refactoring. Proposed methods by Aniche, Mauricio, et al. [2] have shown how different machine learning algorithms can be used to predict refactoring opportunities with a training set of 11,149 real-world projects from the Apache, F-Droid, and GitHub ecosystems and how the random forest classifier gave maximum accuracy out of 6 algorithms to predict method-level, class-level and variable-level refactoring after considering metrics and context of commit.

Upon a new request to add a feature, developers try to decide the refactoring to improve source code maintainability, comprehensibility, and prepare their systems to adapt this new requirement, however this process is difficult and time consuming. Machine learning based approach is a good solution to solve this problem, models trained on history of the previously requested features, applied refactoring, and code smells information outperformed and gave promising results (83.19% accuracy) with 55 open source Java projects [3]. This study aimed to use code smell information, after predicting the need of refcating. Binary classifiers give the need of refactoring and later based on request code smell information along with features is used to predict the refactoring type.The model trained with code smell information lead to the best accuracy.

## 3 Methodology

#### 3.0.1 Overview

Refactoring is necessary to improve the software quality, developers often perform refactoring to maintain their softwares, add new features, and fix problems with existing one. While performing these changes, it is quite difficult to identify the correct refactoring. The methods described in this paper can help developers, maintenance teams to decide necessary refactoring for their software. We have implemented multiple machine learning modes to predict the correct refactoring type based on the commits of project and code metrics. Our approach will also help the development team to decide if any commit is unusual. Detecting anomalies in commits was necessary since we are in the era of the open source community. The models built for this study are based on two approaches, commit message based and code metrics based. After performing some initial analysis we found that Code metrics based models were giving better accuracy as compared to the commit messages. Since, code metrics are the key factors in deciding the cohesion, coupling and complexity of refactoring class. Following section will discuss the methodology we followed to collect the data, preprocess it and build various ML models. As depicted in figure 1, we followed 5 layer architecture to build these models, data collection layer, data preprocessing layer, feature extraction layer model building and evaluation layer. In the data collection layer, we collected commits for projects from github with web crawling, for every project, we prepared csv files with project commits and code metrics for further machine learning analysis. After this initial collection process, data was preprocessed to remove noise from it for model building. Extracting features helped us achieve results, since we were dealing with text data, it was necessary to convert it with useful feature engineering. Preprocessed data with useful features was used for training arious supervised learning models and LSTM models, since it allows us to combine different layers such as text and numeric features. We splitted our analysis in two parts based on our initial experiments. Only commit messages were quite not robust to predict the refcatring type, thus we tried code metrics and combination of commit messages and code metrics. Following section will briefly describe the procedure used to build models with these three inputs.

#### 3.0.2 Overall framework of study with 4 layers

1. Data Collection layer:
2. Data Preprocessing Layer:
   After importing data as pandas dataframes, data is checked for duplicate commit Ids and missing fields. To achieve better accuracy data with duplicate values and missing values should not get considered for further analysis. We also normalized the metric values using standard deviation, randomized the data set with random sampling, removed null entries.
   Text Preprocessing:
   Since we are dealing with commit messages from VCS, text preprocessing is a crucial step.For commit messages to be classified properly by the classifier, they need to be preprocessed and cleaned, put in a format that an algorithm can process. To extract keywords, We have followed the steps listed below:

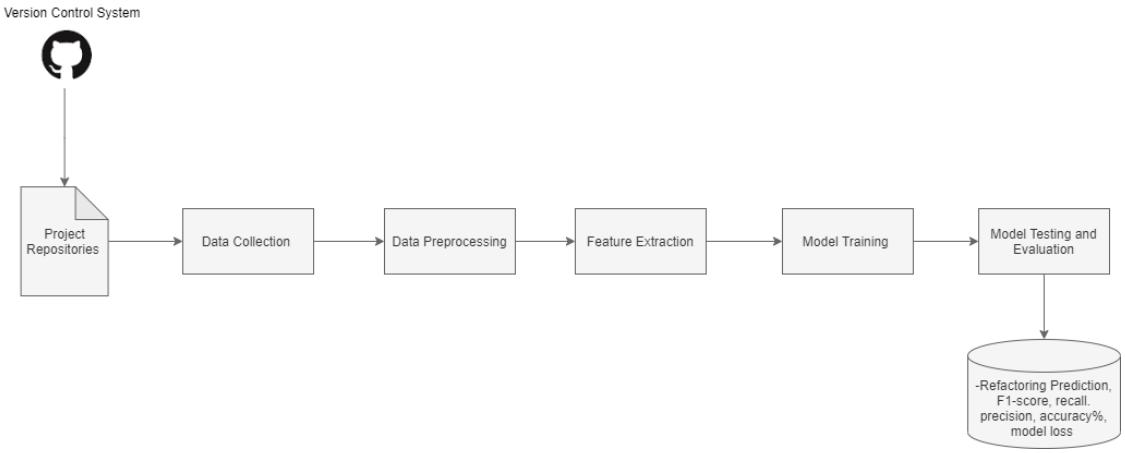


**Fig. 1.** Overall Framework of Model

1.2.1 Tokenization: For text processing, I am using NLTK library from python. The tokenization process breaks a text into words, phrases, symbols, or other meaningful elements called tokens. Here tokenization is used to split commit text into its constituent set of words.
1.2.2 Lemmatization: The lemmatization process replaces the suffix of a word or removes the suffix of a word to get the basic word form. In this case of text processing, lemmatization is used for part of speech identification and sentence separation, keyphrase extraction. Lemmatization gave the most probable form of a word.Lemmatization considers morphological analysis of words this was one of the reason of choosing it over stemming, since stemming only work by cutting off the end or the beginning of the word and takes list of common prefixes and suffixes of it by considering morphological variants, sometimes this might not give us the proper results where sophisticated stemming is required, giving rise to other methodologies such as porter, snowball stemming. This is one of the limitations of stemming method.
1.2.3 Stop Word Removal: Further text is processed for English stop words removal.
1.2.4 Noise Removal: Since data is coming from the web it is mandatory to clean HTML tags from data. Data is checked for special characters, numbers, punctuations for the accurate classifier.
1.2.5 Normalization: Text is normalized, converted all into lowercase for further processing and to remove the diversity of capitalization in text.

3. Feature Extraction: Feature extraction includes extracting keywords from commits, these extracted features are used to build a training data set. For feature extraction we have used a word embedding library from Keras, which gives the indexes for each word. Word embedding helps to extract information from the pattern and occurrences of words. It is an advanced method that goes beyond traditional feature extraction methods from NLP to decode the meaning of words giving more relevant features to our model for training. Word embedding is represented by a single n-dimensional vector where similar words occupy the same vector. To accomplish this, we have used pretrained GloVe word embedding. GloVeword embedding technique is efficient since the vectors generated by using this technique are small in size and none of the indexes generated are empty reducing the curse of dimensionality, whereas other feature extraction techniques such as n-grams, TF-IDF, bag of words generates very huge feature vectors with sparsity which cause memory wastage and increases complexity of algorithm. Steps followed to convert text into word embedding:
3.1.We converted the text into vector, using tokenizer function from keras, then converted sentences into numeric counterparts, applied padding to the commit messages with shorter length
3.2.once we have the padded number sequence representing our commit message corpus. We compared it with the pretrained GloVe word embedding and created the embedding matrix that has words from commit and respective values for each from GloVe embedding. After these steps we have word embedding for all words in our corpus of commit messages.
4. Data Preparation Layer: After preprocessing data will be divided into two sets, one for training purpose and one for testing. We divided 70% data as training dataset and 30% as a testing dataset. Since the number of commits to classify is large, it is difficult to manually process them all, so there is a need to randomly sample a subset while making sure it equitably represents the featured classes for the multiclass classifier
5. Data Modeling and Analysis Layer: After preparing the training dataset, initially we build modes with commit messages as input to predict the refactoring class, we used different supervised learning modes, however poor accuracy lea up to change the direction of experiments and we considered code metrics as well to build model,and

| Parameters used in LSTM Model | Values |
|---|---|
| Number of neurons | 6 |
| Activation Function | softmax |
| Loss Function | categorical_crossentropy |
| Optimizer | adam |
| Number of dense layers | 1 |
| Epoch | 5 |

**Table 1.** Parameter hypertuning for LSTM model

combination of code metrics, commit messages. Following section will describe how we approached this problem of refactoring prediction by considering different sets of inputs.

### 3.0.3 Model with Commit message as input:

Model building and training: To build the model with commit messages as input to predict the refactoring type, we used Keras functional API after we got the word embedding matrix.

We followed the below steps:

1.We created a model with an input layer of word embedding matrix, LSTM layer which gave us a final dense layer of output. 2.For LSTM layer, we used 128 neurons, for dense layer, we have 5 neurons since there are five different refactoring classes 3.We have softmax as an activation function in the dense layer and categorical_crossentropy as loss function. We also performed the parameter hyertuning to choose the values of activation function, optimizer, loss function, number of nodes, hidden layers, epoch,number of dense layers, etc. The dataset and source code of this experiments is available on GitHub [13]. 4.We trained this model on 70% of data with 10 epochs. 5.After checking the training accuracy and validation accuracy we observed this model is not overfitting.

Model testing: To test the model with only commit messages as input we used 30% of data, we used the evaluate function from keras API to test the model on test dataset, and visualized the model accuracy and model loss.

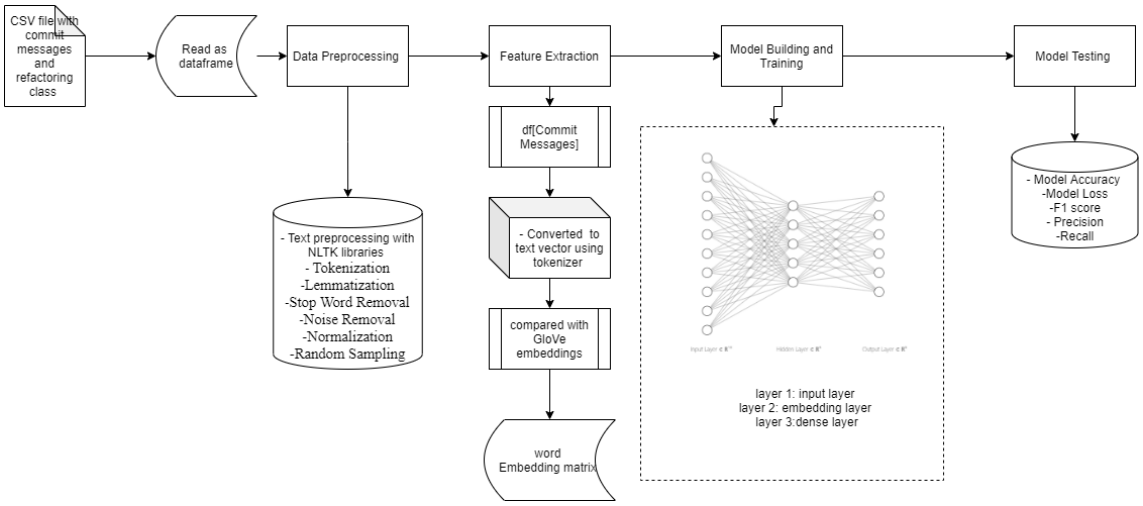


**Fig. 2.** framework of model with commit messages as input *[Eman: We need diagram that white box the selected model like the example we discussed earlier. An example can be found here: https://dl.acm.org/doi/pdf/10.1145/3387904.3389263]*

### 3.0.4 Models with Code metrics as a input:

Model building and training:

After we split the data as training and test dataset. We build different supervised machine learning models to predict the refactoring class. The steps we followed are:

1.We applied SMOTE techniques to balance our dataset, since this dataset was highly imbalanced. 2.We used supervised machine learning models from sklearn library of python 3.We trained random forest classifier with balanced

| Supervised machine learning model | Parameters | Values |
|---|---|---|
| SVM | C | 1.0 |
| | Kernel | Linear |
| | Gamma | auto |
| | Degree | 3 |
| Random Forest | n_estimators | 100 |
| | criterion | gini |
| | min_samples_split | 2 |
| Logistic Regression | penalty | 12 |
| | dual | False |
| | tol | 1e-4 |
| | C | 1.0 |
| | fit_intercept | True |
| | solver | lbfgs |
| Naive Bayes | alpha | 1.0 |
| | fit_prior | True |
| | class_prior | None |

**Table 2.** Parameter hypertuning for Supervised ML algorithms

class We also trained SVM and Logistic regression classifiers on 70% of data. 4. For SVM model, we performed the parameter hypertuning to get optimal results. The below table shows the selected parameters for each algorithms used in this experiment.

Model testing: Built models are tested on 30% of data, and results were analyzed by varied machine learning measures such as precision, recall, F1- score, accuracy, confusion matrix,etc.

1.Precision: it gives us more insights about model performance by showing how many of the positive predictions made are correct. 2.Recall: recall gives us the ratio of correctly classified positives to the actual positives from the data. 3.F-1 Score: we aimed to get a good f-1 score since this measure gives us a weighted average of precision and recall.

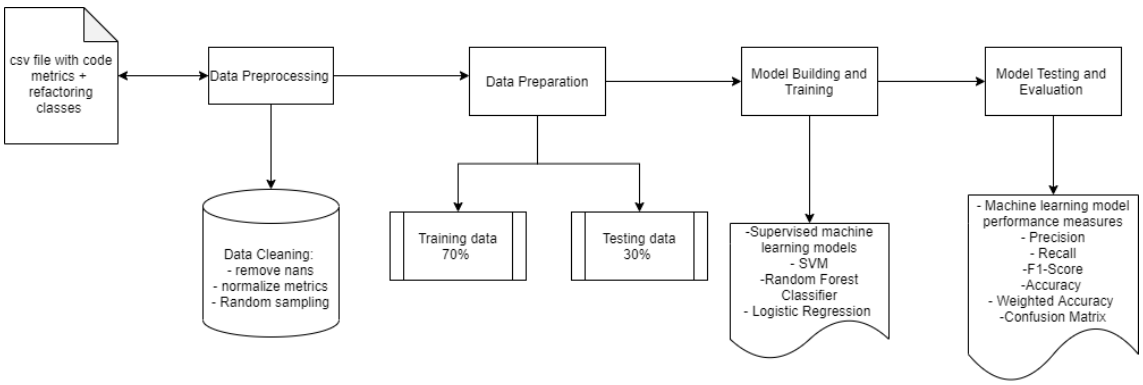


**Fig. 3.** framework of model with code metrics as input *[Eman: We need diagram that white box the selected model like the example we discussed earlier]*

### 3.0.5 Models with commit message and code metrics as input:

Model building and training:

To build a deep learning model with a multi input we used a Keras functional API and followed the below steps: 1.Here, we first build a submodel to accept inputs as a commit message, we build word embeddings for the commits messages. 2.This submodel has an input layer, embedding layer and a LSTM layer of 128 neurons. Second submodel will deal with the code metrics. This second submodel of code metrics has 3 layers, input layer and 2 dense layers. 3.The output from the LSTM layer of the first submodel concatenated with output from the dense layer of the second model gave us the input layer for the final model. 4.This layer is concatenated with another 10 neurons and a dense layer of 5 neurons since we have 5 refactoring classes giving us our final LSTM model. 5.For each submodel, we performed the

| Parameters used in LSTM submodels | Values |
|---|---|
| Number of neurons | 10 |
| Activation Function | softmax |
| Loss Function | $\text{categorical}_c rossentropy$ |
| Optimizer | adam |
| Number of dense layers | 4 |
| Epoch | 10 |

**Table 3.** Parameter hypertuning for concatenated LSTM model

parameter hyertuning to choose the values of activation function, optimizer, loss function, number of nodes, hidden layers, epoch,number of dense layers, etc.

Model testing: To test the model with only commit messages as input we used 30% of data, we used the evaluate function from keras API to test the model on test dataset, and visualized the model accuracy and model loss.

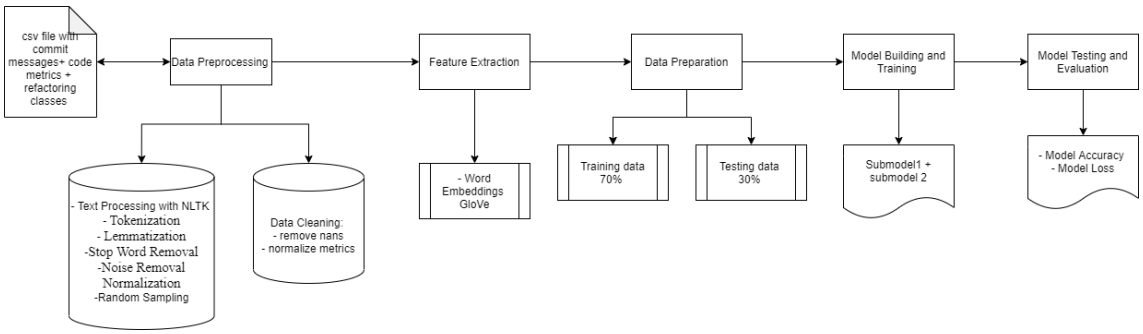


**Fig. 4.** framework of model with commit messages + code metrics as input *[Eman: We need diagram that white box the model like the example we discussed earlier]*

## 4 Experimental Results and Analysis

The following section will describe the experimental setup and the results obtained, followed by the analysis of research questions. The study done in this paper can also be extended in future to identify usual and unusual commits. Building multiple models with combinations of input gave us better insight of factors impacting refactoring class prediction.

### 4.1 Dataset

Dataset 1: commit messages and refactoring classes

We are using a dataset with 5 different refactoring classes, rename, push down, inline, extract and pull up. The dataset used for this experiment is quite balanced. There are a total 3314 commits in this dataset.

| Refactoring Classes | Count |
|---|---|
| rename | 834 |
| Push down | 834 |
| inline | 834 |
| extract | 834 |
| Pull up | 834 |

Commit Messages : Examples of commit messages for each refactoring class:

rename-

Lifecycle related changes * renamed the stages (StartCluster ¿ ServiceStart *Kill* ¿ *Stop*) and updates properties according to that * renamed (Start—Stop)* ¿ *(Start—Stop) * added lifecycle listeners registered on SlaveState

Push down-

improved handshake classes ( initiated by eab25a4 and 7c84fb6)

inline-

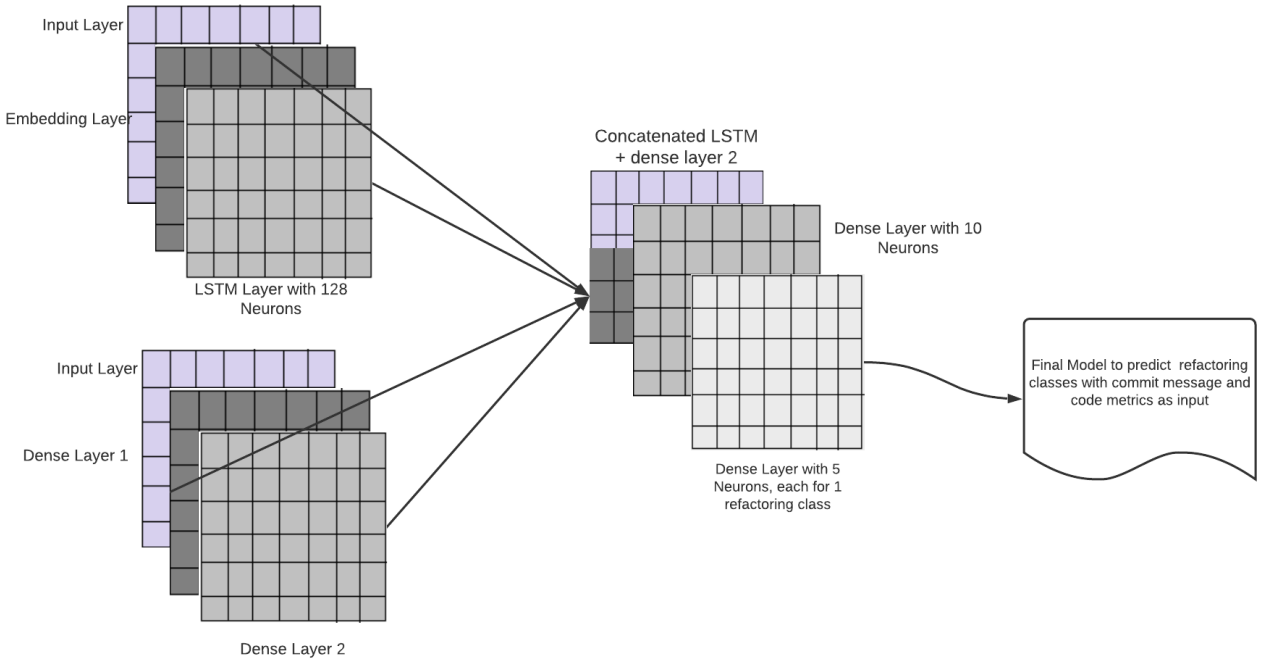


**Fig. 5.** final LSTM model with code metrics and commit messages submodels *[Eman: We need another diagram to show the internal structure of the model (whitebox): done]*

PC optimization refactoring (Only unit tests. Missing IT tests refactoring.)
extract-
Rework of WampMessage serialization. Added binary message using Message Pack.
Pull up-
Cleaned up indexing entry points and documentation. git svn id: svn+ssh://svn.internal.sanger.ac.uk/repos/svn/pathsoft/psu ee4ac58c ac51 4696 9907 e4b3aa274f04

Dataset 2: code metrics and refactoring classes:
Our dataset has 64 code metrics as features such as 6 refactoring classes, rename, pull up, push down, extract, move and inline. This dataset is imbalanced , and thus we have used SMOTE technique while training the model.

| Refactoring Classes | Count |
|---|---|
| rename | 1253 |
| Push down | 372 |
| inline | 122 |
| extract | 333 |
| Pull up | 1018 |
| Move | 215 |

### 4.2 Experiment with commit messages as input:

In this experiment, we followed basic text classification where we assigned a class label to each commit based on a trained classification model, the model is trained on only commit messages to predict refactoring classes. The dataset

used to perform this experiment was very balanced since it has an equal number of instances for each class. To begin with we first imported our collected csv data file, performed basic analysis.

We followed the below steps:

- Imported needed python libraries for this experiment, since dealing with text data, we need all NLTK libraries from Keras.
- Normalized and randomized the dataset for better accuracy.
- Performed the text preprocessing, tokenization, lemmatization, stop word removal,etc.
- Feature extraction to build word embeddings.
- Trained the LSTM model on 10 epochs.
- Tested on 30% of dataset.

Results:

| | |
|---|---|
| Model Accuracy | 54.3% |
| Model Loss | 1.401 |
| f1-score | 0.21035261452198029 |
| Precision | 1.0 |
| Recall | 0.1176215410232544 |

**Table 4.** Results of LSTM model with commit messages as input

Results per class:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| extract | 0.56 | 0.66 | 0.61 | 92 |
| inline | 0.47 | 0.43 | 0.45 | 84 |
| rename | 0.56 | 0.68 | 0.62 | 76 |
| push down | 0.37 | 0.39 | 0.38 | 87 |
| pull up | 0.41 | 0.27 | 0.32 | 89 |
| move | 0.97 | 0.95 | 0.96 | 73 |
| accuracy | | | 0.55 | 501 |
| macro avg | 0.56 | 0.56 | 0.56 | 501 |
| weighted avg | 0.54 | 0.55 | 0.54 | 501 |

**Table 5.** Metrics per class

This model gave a total of 54% accuracy on 30% of test data. With the 'evaluate' function from keras, we were able to evaluate this model. The overall accuracy and model loss shows that only commit messages is not very robust input to predict refactoring class, there are a number of reasons why the commit messages are unable to build robust predictive models. Often dealing with text to build a classification model is challenging, feature extraction helped us to achieve this accuracy. Most of the time the use of limited vocabulary by developers makes commits unclear and difficult to follow for fellow developers. In text classification, initial class labelling is quite complex and impacts the overall accuracy of the model. This was the one of the reasons why we continued researching in the same area and tried different sets of inputs to achieve good accuracy. As you can see from below graphs, this model has an overfitting problem.

### 4.3 Experiment with code metrics as input:

To perform this experiment, we considered source code metrics as input for our classification model, there are 64 code metrics in our dataset, source code metrics helps to understand cohesion, coupling and leads to more accurate prediction of refactoring class, since these metrics are closely related to each other. Metrics also helps to identify which refactoring class is suitable for changes made in the source code, and leverages traceability. We dealt with only numbers

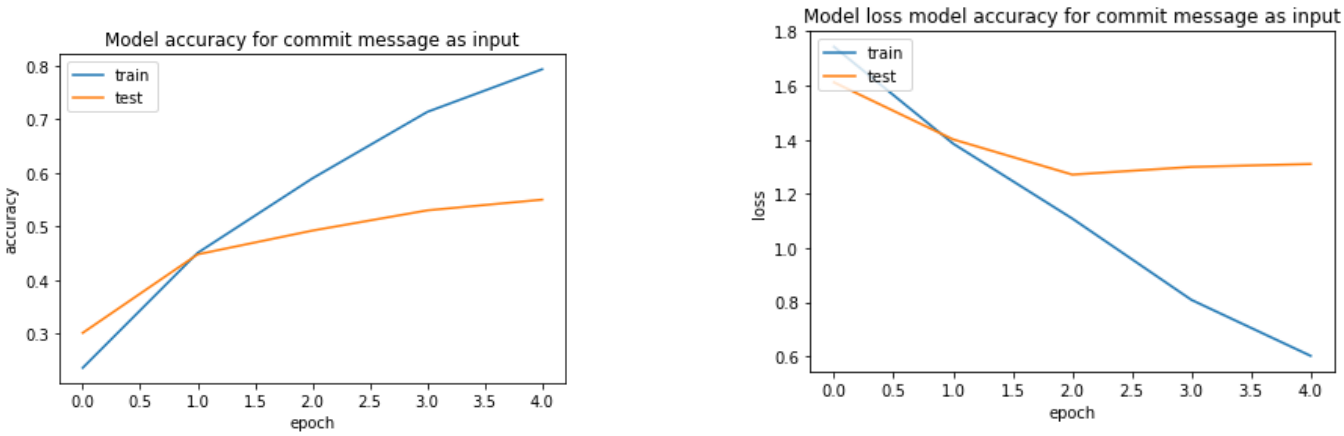


**Fig. 6.** Model Accuracy and Loss for commit message as input

while performing this experiment. We trained different supervised machine learning models to predict refactoring class based on code metrics. We initially analyzed this dataset, and carried out exploratory data analysis.
To accomplish this task we followed the below steps:

- Imported the data in jupyter.
- Removed null entries, performed random sampling, normalized the data using statistical modeling.
- Since the collected data set was highly imbalanced we used Synthetic Minority Oversampling Technique(SMOTE) technique to balance it before training the model.
- Imbalanced dataset gives very poor performance, SMOTE technique over samples the minority class by duplicating the examples from the same class. It synthesizes new examples from existing ones. This helped us to achieve better results.
- With a SMOTE-transformed training dataset we trained four different models, random forest classification model, Support Vector Machine model, Naive bayes and Logistic regression model to predict the refactoring class
- Tested all models on 30% of data, results obtained were surprising , only random forest classification model gave us good results amongst four.

Random Forest Multiclass Classification: We used the inbuilt functions from scikit-learn library of python to build a random forest model and to test it. We followed the below approach to set up this experiment:

- Imported pandas, numpy and sklearn library from python
- Imported dataset as data frame
- Removed null entries,randomized the dataset and normalized it with statistical modeling.
- Trained the random forest model and tested it.

This model gave us overall 75% accuracy, and this was the best accuracy we got so far. We used precision, recall, F1-score and support as measuring units to evaluate the model. Precision, recall, F-1 score for this model is shown in the below table of results. Precision gives the percentage of positive predictions made correctly, whereas recall depicts the percentage of positive instances predicted correctly by classifier over all positive instances in the data set. F-1 score is nothing but a harmonic mean of precision and recall.

Logistic Regression:
Logistic regression is a predictive analysis algorithm based on the concept of probability and it uses sigmoid function to predict the classes. We have used the in built functions from the sklearn library of python to achieve the results. As you can see from the below results table, this model achieved only 47% accuracy, with best class accuracy for inline class.

SVM:
SVM is one of the robust algorithms for multiclass classification due to its ability to not overfit models with multiple classes. SVM predicts the classes based on the separation distance in the hyperplane of data points, called support vectors. For this experiment we have used linear kernel function. The following results table shows the overall accuracy and class accuracies. This model achieved 44% of accuracy with best class accuracy for inline class of 79%.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| extract | 0.91 | 0.47 | 0.62 | 395 |
| inline | 0.99 | 1.00 | 1.00 | 422 |
| move | 0.94 | 0.69 | 0.80 | 429 |
| pull up | 0.57 | 0.65 | 0.61 | 413 |
| push down | 0.90 | 0.69 | 0.78 | 373 |
| rename | 0.56 | 0.98 | 0.71 | 449 |
| accuracy | | | 0.75 | 2481 |
| macro avg | 0.81 | 0.75 | 0.75 | 2481 |
| weighted avg | 0.81 | 0.75 | 0.75 | 2481 |

**Table 6.** Results of Random Forest algorithms

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| extract | 0.56 | 0.29 | 0.38 | 380 |
| inline | 0.79 | 0.72 | 0.76 | 417 |
| move | 0.94 | 0.69 | 0.80 | 429 |
| pull up | 0.41 | 0.13 | 0.20 | 418 |
| push down | 0.62 | 0.40 | 0.48 | 400 |
| rename | 0.32 | 0.90 | 0.47 | 443 |
| accuracy | | | 0.47 | 2481 |
| macro avg | 0.53 | 0.46 | 0.45 | 2481 |
| weighted avg | 0.53 | 0.47 | 0.45 | 2481 |

**Table 7.** Results of Logistic Regression model

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| extract | 0.64 | 0.24 | 0.35 | 380 |
| inline | 0.79 | 0.73 | 0.76 | 417 |
| move | 0.69 | 0.30 | 0.42 | 423 |
| pull up | 0.29 | 0.11 | 0.16 | 418 |
| push down | 0.61 | 0.29 | 0.40 | 400 |
| rename | 0.29 | 0.92 | 0.44 | 443 |
| accuracy | | | 0.44 | 2481 |
| macro avg | 0.55 | 0.43 | 0.42 | 2481 |
| weighted avg | 0.55 | 0.44 | 0.42 | 2481 |

**Table 8.** Results of SVM model

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| extract | 0.20 | 0.73 | 0.31 | 380 |
| inline | 0.65 | 0.60 | 0.62 | 417 |
| move | 0.39 | 0.25 | 0.31 | 423 |
| pull up | 0.47 | 0.13 | 0.20 | 418 |
| push down | 0.50 | 0.34 | 0.41 | 400 |
| rename | 0.72 | 0.08 | 0.14 | 443 |
| accuracy | | | 0.35 | 2481 |
| macro avg | 0.49 | 0.35 | 0.33 | 24811 |
| weighted avg | 0.49 | 0.35 | 0.33 | 2481 |

**Table 9.** Results of Naive bayes model

Naive bayes:

Naive bayes is one of the simplest classification algorithms but this algorithm has few cons when we are dealing with multiclass classification. It only works best with categorical data. It takes the freicy class occurrence with correlation with other classes into consideration. From the result's table, it is clear that naive bayes is not very robust for code metrics, this model was only 35% accurate in predicting refactoring class.

### 4.4 Experiment with commit messages and code metrics as input:

This experiment was accomplished by building a multilayer LSTM model to predict refactoring class. Commit message and 64 code metrics for each commit served as one set of input for every instance. We combined the dataset of commit messages with code metrics for this experiment. Combining text input with numbers was challenging, we first performed the data cleaning and exploratory data analysis followed by feature extraction for commit messages with GloVe word embedding. The vector generated by this is used as input for our first submodel from LSTM, the second model is built with code metrics as input. The output from the LSTM layer of the first submodel concatenated with output from the dense layer of the second model gave us the input layer for the final model. Final model is built with this input and refactoring classes. Model is trained on 70% of data with 10 epochs. Testing the model was little arduous, since we had to combine text and numbers to get a proper testing dataset. Due the overfitting model achieved 40.67% of accuracy and from the graphs of model loss and accuracy, overfitting is noticeable.
Results:

| Model Accuracy | 40.67% |
|---|---|
| Model Loss | 1.310 |
| F-score | 0.014 |
| Precision | 0.666 |
| Recall | 0.0071 |

**Table 10.** Results of LSTM model with commit messages and code metrics as input

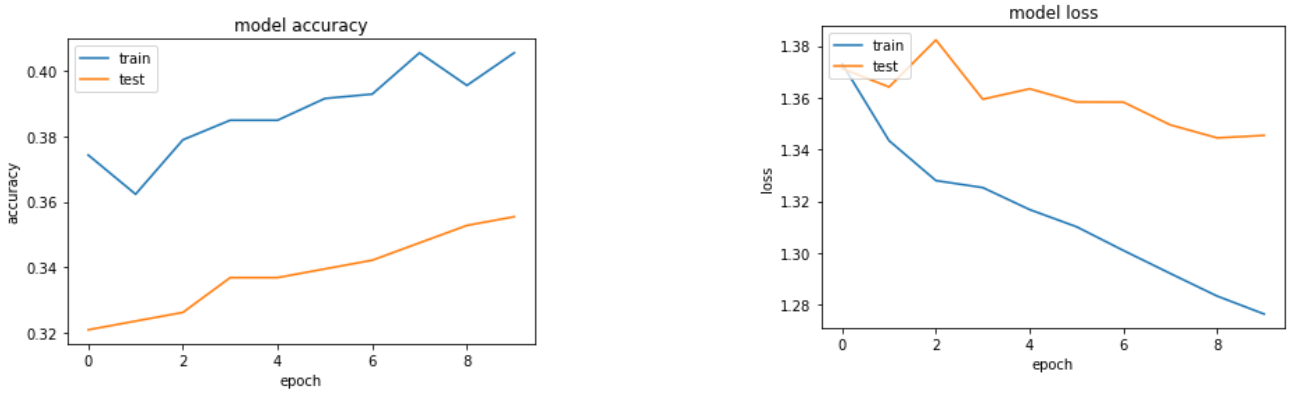


**Fig. 7.** Model Accuracy and Loss for commit message and code Metrics as input

## 5 Research Questions and Limitations

The research questions listed below investigates the results obtained from above all experiments, it also initiates new questions for future research. Questions 1,2 and 3 can be answered with the help of analysis done after obtaining the results.
*[Eman: We need to add RQ about what are the important features per class? to answer the question, we can add a table that lists top 10 features per class: done]*

- RQ1:How accurate are commit messages in predicting refactoring classes?
- RQ2: How accurately commit messages with source code metrics can predict refactoring class for any project?
- RQ3: How does the code metrics influence the refactoring class prediction?
- RQ4: Which refactoring classes were most accurately classified by each method used in this study?

RQ1:How accurate are commit messages in predicting refactoring classes?

One of the very first experiments performed gave us the answer to this question, where we used only commit messages to train the LSTM model to predict refactoring class. The accuracy of this model was 54%, and was not up to the expectations. Thus we concluded that only commit messages are not very effective in predicting refactoring classes, we also noticed that the developers ability to use minimal vocabulary while writing code and committing changes on version control systems could be one of the reasons for inhibited prediction.

RQ2: How accurately commit messages with source code metrics can predict refactoring class for any project?

Combining commit messages with source code metrics resulted in poor prediction accuracy, we built a multilayer LSTM model to predict the refactoring class with commit messages and code metrics as input and this model was only 40% accurate in class prediction. Our experiment supports our conclusion , commit messages with code metrics have little impact on refactoring classes.

RQ3: How does the code metrics influence the refactoring class prediction?

Random forest multiclass classification model built with source code metrics resulted in good accuracy whereas other 3 supervised learning models gave us poor accuracy, in this case our experiments shows random forest classification model is significantly robust in predicting refactoring classes when built on code metrics input.

RQ4: Which refactoring classes were most accurately classified by each method used in this study?

For the study where we considered commit messages as input for the LSTM model, 'move' class was the most accurately predicted. After training supervised learning algorithms with code metrics as input, Random forest algorithm predicted inline class with 99% accuracy, whereas logistic regression predicted move with 94% of accuracy. The overall accuracy of the logistic regression classifier was noticeably poor. SVM and Naive bayes predicted inline, rename refactoring classes more accurately.

#### 5.0.1 Limitations:

Prediction of refactoring class using supervised machine learning algorithms gave us better results. Random forest with SMOTE technique was the most optimal algorithm when we considered code metrics as input to train the model, whereas other two experiments where we trained LSTM model with commit messages and combination of code metrics, commits we got poor accuracy. Developers' way of writing short commits with minimum vocabulary , text data, imbalanced dataset could be few reasons for this poor results. Dealing with text data was challenging, thus we considered a deep learning method for experiments.

We have only considered 6 refactoring classes, inline, extract, pull up, push down, move, rename. In future we can extend the scope of our study by considering more classes. code metrics are the key factors in deciding the cohesion, coupling and complexity of refactoring class, we have considered the dataset with 64 code metrics. We could also extend our research by taking more metrics into account. This will extend the scope of our study. Limited set of code metrics and refactoring classes was one of the limitations of this study. The methods used to deal with text data and numeric data were different, using a deep learning method with code metrics dataset was another option to consider.Another limitation of this study is focusing on only LSTM, In future, we will also test different deep learning algorithms.

*[Eman: please add a limitation section: done]*

## 6 Conclusion and Future Work

In this paper, we implemented different supervised machine learning models and LSTM models to predict the refactoring class for any project. We considered the combination of inputs to build and test the model. To begin with, we implemented a model with only commit messages as input but this approach led us to more research with other inputs, combining commit messages with code metrics was our second experiment, the model built with LSTM gave us 54.3% of accuracy. 64 different code metrics dealing with cohesion, coupling characteristics of code is one of the best performing models giving us 75% of accuracy when tested with 30% of data. Our study significantly proved that code metrics are effective in predicting the refactoring class, since the commit messages with little vocabulary is not sufficient to train ML models. In future, we would like to extend our scope of study and build various models to detect unusual commits based on refactoring class, commit messages, code metrics, we are also interested in knowing how the predicted refactoring class helps to detect unusual commits. Furthermore we would like to build a software to detect unusual commits from projects, we plan to use BERT algorithm to improve the performance of models with commit message as input.

## References


1. S. Zafar, M. Z. Malik, and G. S. Walia, "Towards standardizing and improving classification of bug-fix commits," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–6.
2. M. Aniche, E. Maziero, R. Durelli, and V. Durelli, "The effectiveness of supervised machine learning algorithms in predicting software refactoring," *IEEE Transactions on Software Engineering*, 2020.
3. A. S. Nyamawe, H. Liu, N. Niu, Q. Umer, and Z. Niu, "Feature requests-based recommendation of software refactorings," *Empirical Software Engineering*, vol. 25, no. 5, pp. 4315–4347, 2020.
4. J. He, L. Xu, M. Yan, X. Xia, and Y. Lei, "Duplicate bug report detection using dual-channel convolutional neural networks," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 117–127.
5. S. Levin and A. Yehudai, "Boosting automatic commit classification into maintenance activities by utilizing source code changes," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2017, pp. 97–106.
6. C. Casalnuovo, Y. Suchak, B. Ray, and C. Rubio-González, "Gitcproc: A tool for processing and classifying github commits," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 396–399.
7. S. Gharbi, M. W. Mkaouer, I. Jenhani, and M. B. Messaoud, "On the classification of software change messages using multi-label active learning," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 1760–1767.
8. S. Shekarforoush, R. Green, and R. Dyer, "Classifying commit messages: A case study in resampling techniques," in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 1273–1280.
9. R. Xie, L. Chen, W. Ye, Z. Li, T. Hu, D. Du, and S. Zhang, "Deeplink: A code knowledge graph based deep learning approach for issue-commit link recovery," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 434–444.
10. S. Hönel, M. Ericsson, W. Löwe, and A. Wingkvist, "Importance and aptitude of source code density for commit classification into maintenance activities," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2019, pp. 109–120.
11. A. Mauczka, F. Brosch, C. Schanes, and T. Grechenig, "Dataset of developer-labeled commit messages," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 490–493.
12. R. Goyal, G. Ferreira, C. Kästner, and J. Herbsleb, "Identifying unusual commits on github," *Journal of Software: Evolution and Process*, vol. 30, no. 1, p. e1893, 2018.
13. P. Sagar. Source code of experiment. [Online]. Available: https://github.com/smilevo/refactoring-metrics-prediction