Classification of refactoring types using commit messages

Amulya Reddy Maligireddy [*], Ashini Anantharaman[†], Rahul Gowda Kengeri Kiran[‡], Victor Peterson[§]
Golisano College of Computing and Information Sciences, Rochester Institute of Technology
Rochester, New York
Email: [*]am8735@g.rit.edu, [†]aa9162@g.rit.edu, [‡]rk1087@g.rit.edu, [§]vp4175@g.rit.edu

I. Introduction

Commit messages play a vital role in the refactoring process of software development. Refactoring is a crucial step in maintaining and improving the quality of code, and commit messages serve as a permanent record of the changes made to the source code during this process. Clear and descriptive commit messages provide valuable context for other developers, and help them understand the purpose and impact of the refactoring. By providing a clear and concise explanation of the refactoring, commit messages can greatly improve the maintainability of the code and foster effective collaboration within a development team. However, despite their importance, the way in which developers write commit messages for refactoring is not well understood, and it remains uncertain whether these messages accurately reflect the type of refactoring.

II. Existing Studies

There has been a significant amount of research focused on identifying and detecting refactoring activities in the software development process. One common approach to this is the analysis of commit messages in versioned repositories. Stroggylos and Spinellis [1] used a keyword-based approach that searched for words stemming from the verb "refactor," such as "refactoring" or "refactored." Similarly, Ratzinger et al. [2] and Ratzinger [3] used a set of 13 keywords, including "refactor," "restruct," and "clean," to identify refactoring activities in a pair of program versions. However, the reliability of commit messages in indicating refactoring activities has been called into question by several studies. Murphy-Hill et al. [4] replicated Ratzinger's experiment in two open-source systems and found that commit messages are unreliable indicators of refactoring activities due to the inconsistent documentation of these activities. In contrast, a study by Soares et al. [5] found that manual analysis was the most reliable approach in detecting behavior-preserving transformations, outperforming both commit message analysis and dynamic analysis. On the other hand, a survey by Kim et al. [6] found that only 5.76% of commits included refactoring-related keywords, with 94.29% of commits lacking these keywords. These studies suggest that although there are various structural metrics that can represent internal quality attributes, not all of them can measure what developers consider to be an improvement in their source code.

III. Problem Statement

In the paper [7], the prediction of refactoring were formulated as a multi-class classification problem, i.e., classifying refactoring commits into six method-level refactoring operations. From the data file provided Message.csv, we understand that there are 361 duplicate commit messages out of 5004 entries, which means for some commit messages there are multiple refactoring methods resulting in ambiguity. To tackle this problem, we will have to make the classification multi-labeled.

IV. Proposed Solution

The goal of the project is to automatically identify the refactoring type using the commit messages provided. We are planning to replicate the procedures followed in [7]. So we will be mainly focusing on Method level refactoring: Extract Method, Inline Method, Move Method, Pull-up Method, Push-down Method, and Rename Method. There are different steps involved in classification of refactoring types starting from data collection and refactoring detection, data labeling, text cleaning and pre-processing, feature extraction using N-Gram, model training and building and model evaluation. In the data extraction phase, we are planning to use project which are within recent time and, we will target only the non-forked projects to avoid any duplication of data. The data labeling will be done by using multi-label classification instead of multi-class classification where one commit message can belong to multiple refactoring types. In the text cleaning and pre-processing different processes like Tokenization, Lemmatization, Stop-Word removal, Noise removal and Capitalization Normalization will be done. Then in the Feature Extraction using N-gram, bigrams will be used to calculate the TF-IDF scores. After all the cleaning and preprocessing we will be using top classifiers like K-Nearest Neighbor (KNN), Naive Bayes Multinomial (NBM), Gradient Boosting Machine (GBM), and Random Forest (RF) and conduct a model evaluation.

V. Study Design

We are trying to replicate the work done by the authors in the paper [7]. We were provided with message.csv file containing data about the commit messages and the labels for the type of refactoring methods.

Our solution design has four main phases: (1) text cleaning and preprocessing, (2) feature extraction using N-Gram, (3) model training and building, and (4) model evaluation.

1) **Text cleaning and preprocessing**: In order for the commit messages to be classifed into correct categories, they need to be preprocessed and cleaned; put into a format that the classifcation algorithms will process. This way, the noise will be removed, allowing for informative featurization. To extract features (i.e., words), we preprocess the text as follows:

   A. Tokenization

   The goal of tokenization is to investigate the words in a sentence. The tokenization process breaks a stream of text into words, phrases, symbols, or other meaningful elements called tokens. We tokenize each commit by splitting the text into its constituent set of words. We also split tokens on special characters (e.g., the string ”show_bug” would be separated into two tokens, ”show” and ”bug”).

   B. Lemmatization

   Lemmatization is the process of grouping together the inflected forms of a word so they can be analysed as a single item, identified by the word's lemma. In our work, we created dictionary form by disassembling sentences into their component components of speech. Given that the input information could consist of a lengthy section of text, we divided the commit messages into sentences. The lemmatization process either modifies or removes a word's suffix to produce the core word form (lemma). Key-phrase extraction is aided by using filtering words as traits.

   C. Stop-word removal

   Stop words, i.e., words and common English terms like ”is,” ”are,” ”if,” etc., are eliminated because they serve no purpose as classifier features.

   D. Capitalization normalization

   Because text may employ a variety of capitals to create sentences, which could provide a problem when classifying large commits, all words in the commit messages are turned to lower case, and all verb contractions are expanded.

   E. Noise removal

   Special letters and numbers are removed to protect the categorization. We carefully reject URLs, email addresses, unusual characters, and characters that are both unique and repeated.

2) **Feature extraction using N-Gram**: Following text cleaning and preprocessing, feature extraction is used to extract only the most pertinent data from text strings in order to distinguish between classes in both classification problems.We specifically used the Bi-Gram feature extraction method. The value of a Bi-Gram is calculated by dividing its TF score by its IDF score. As a result, a value is assigned to each preprocessed word in the commit message that represents the word's weight according to this weighting approach. The TF-IDF values terms that occur frequently in fewer texts more highly than terms that do so frequently in a large number of texts.

3) **Model training and building**: We split the data into train and test sets in 80 to 20 ratio. We have also performed ten fold cross-validation technique to assess the variability and reliability of the classifier. We did not see much of a difference between both approaches. But in future work, we would emphasize on K fold cross validation technique.

4) **Model evaluation**: We have considered the following classifiers: Random Forest, Logistic regression and Gradient Boosted machine. We have trained the model with training data and will be testing the model with the commit messages across and the output would be evaluated We investigated each classifier in our study using common statistical measures Precision, Recall, and F-measure of classification performance.

VI. Experimentation

This section aims to investigate the effectiveness of our proposed approach for predicting the type of refactoring. To this end, we designed a series of experiments to answer the following three research questions:

- *RQ1: How effective is our supervised learning in predicting the type of refactoring?*
- *RQ2: How does our model compare to the results presented by the authors' model?*
- *RQ3: What are the frequent terms utilized by developers per refactoring type, and how does that compare to our model's predicted terms?*

The results of our experiments and a detailed analysis of the findings are provided. The following subsections provide a detailed account of our approach and the experiments conducted to answer each research question.

A. RQ1. How effective is our supervised learning in predicting the type of refactoring?

The below images show the performance results of each classifier, in terms of precision, recall and F-measure, broken down per class i.e, refactoring types (extract, inline, move, pull up, push down, rename).

Our research focuses on the three top-performing models discussed by the authors of the paper [7], namely Random

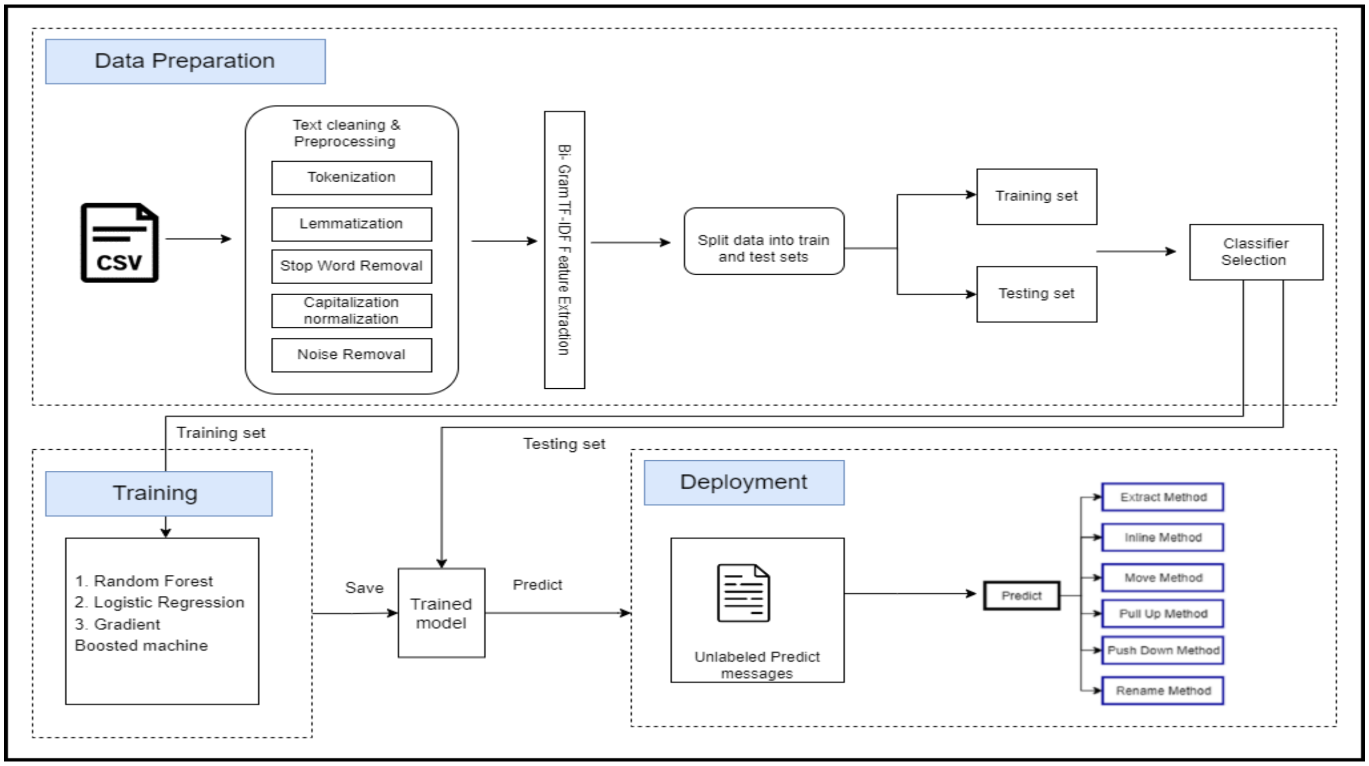
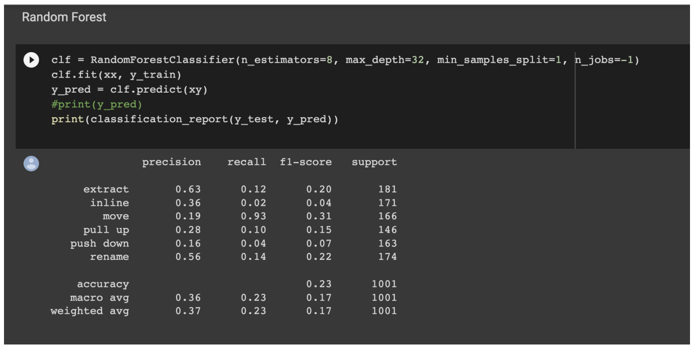


Fig. 1. Overall Prediction Architecture


Random Forest

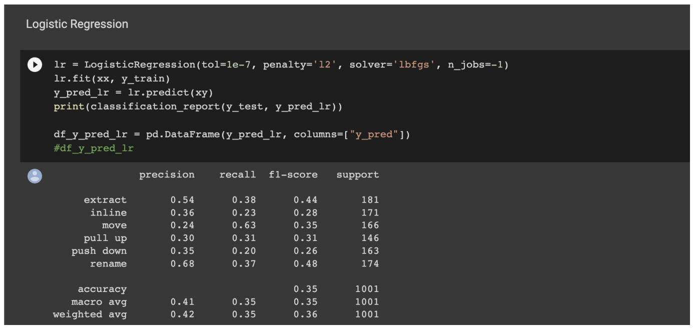
```
clf = RandomForestClassifier(n_estimators=8, max_depth=32, min_samples_split=1, n_jobs=-1)
clf.fit(xx, y_train)
y_pred = clf.predict(xy)
#print(y_pred)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

     extract       0.63      0.12      0.20       181
      inline       0.36      0.02      0.04       171
        move       0.19      0.93      0.31       166
     pull up       0.28      0.10      0.15       146
   push down       0.16      0.04      0.07       163
      rename       0.56      0.14      0.22       174

    accuracy                           0.23      1001
   macro avg       0.36      0.23      0.17      1001
weighted avg       0.37      0.23      0.17      1001
```

Fig. 2. Classification Report for Random Forest


Logistic Regression

```
lr = LogisticRegression(tol=1e-7, penalty='l2', solver='lbfgs', n_jobs=-1)
lr.fit(xx, y_train)
y_pred_lr = lr.predict(xy)
print(classification_report(y_test, y_pred_lr))

df_y_pred_lr = pd.DataFrame(y_pred_lr, columns=["y_pred"])
#df_y_pred_lr
```

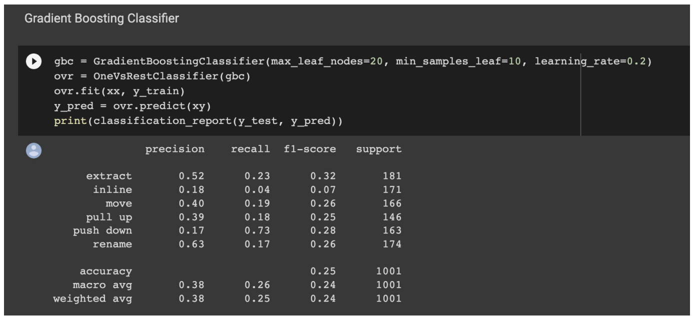
```
              precision    recall  f1-score   support

     extract       0.54      0.38      0.44       181
      inline       0.36      0.23      0.28       171
        move       0.24      0.63      0.35       166
     pull up       0.30      0.31      0.31       146
   push down       0.35      0.20      0.26       163
      rename       0.68      0.37      0.48       174

    accuracy                           0.35      1001
   macro avg       0.41      0.35      0.35      1001
weighted avg       0.42      0.35      0.36      1001
```

Fig. 3. Classification Report for Logistic Regression

Forest Classifier, Gradient Boosted Classifier, and Logistic Re-


Gradient Boosting Classifier

```
gbc = GradientBoostingClassifier(max_leaf_nodes=20, min_samples_leaf=10, learning_rate=0.2)
ovr = OneVsRestClassifier(gbc)
ovr.fit(xx, y_train)
y_pred = ovr.predict(xy)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

     extract       0.52      0.23      0.32       181
      inline       0.18      0.04      0.07       171
        move       0.40      0.19      0.26       166
     pull up       0.39      0.18      0.25       146
   push down       0.17      0.73      0.28       163
      rename       0.63      0.17      0.26       174

    accuracy                           0.25      1001
   macro avg       0.38      0.26      0.24      1001
weighted avg       0.38      0.25      0.24      1001
```

Fig. 4. Classification Report for Gradient Boosted Classifier

gression. We observe that Logistic Regression has an average F-measure of 0.36, when compared to Random Forest and Gradient Boosting, which had F-measures of 0.17 and 0.24 respectively. Logistic Regression also has the highest precision and recall scores for the extract and rename refactoring method categories, indicating that it correctly identified these categories more often than the other algorithms. Overall, we can observe from the results that the rename refactoring types is the easiest to classify with F1-Measures ranging from 0.68 (Logistic Regression) to 0.56 (Random Forest). The extract type is the second easiest type to classify with F1-Measures ranging from 0.63 (Random Forest) to 0.52 (Gradient Boosting Classifier). Inline, Pull-Up, and Push-Down methods were the most difficult to classify by all the algorithms, with Random Forest scoring 0.04 for push-down method. Overall, it appears that more work needs to be done to improve the effectiveness of the models in predicting the type of refactoring. This may involve exploring different features or feature engineering techniques, trying different algorithms, or tuning hyper parameters of the models.

B. RQ2. How does our model compare to the results presented by the authors' model?

We conducted an evaluation of our approach and the authors' approach using precision, recall, and F-measure metrics for the actions of Extract, Inline, Move, Pull up, Pull down, and Rename. To ensure a fair comparison, we compared the performance of our Logistic Regression model with the Logistic Regression model presented by the authors.

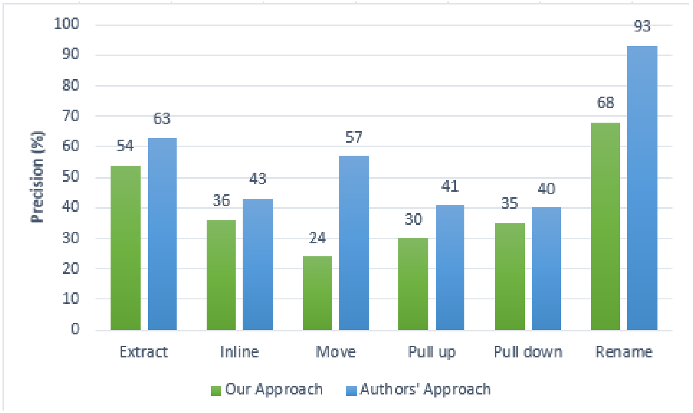


Fig. 5. Visualization of the precision for different approaches

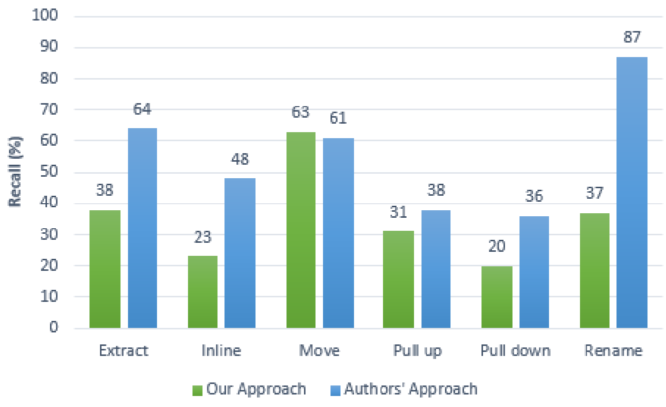


Fig. 6. Visualization of the recall for different approaches

The experimental results presented in Fig 5, Fig 6, and Fig 7 shows that the performance of both approaches varied across different actions. Our approach achieved similar precision but lower recall and F-measure than the authors' approach for the action of Extract. Similarly, for the action of Inline, our approach achieved similar precision but lower recall and F-measure. For the action of Move, our approach had a higher recall but lower precision and F-measure. In the case of Pull up, our approach achieved similar recall but lower precision

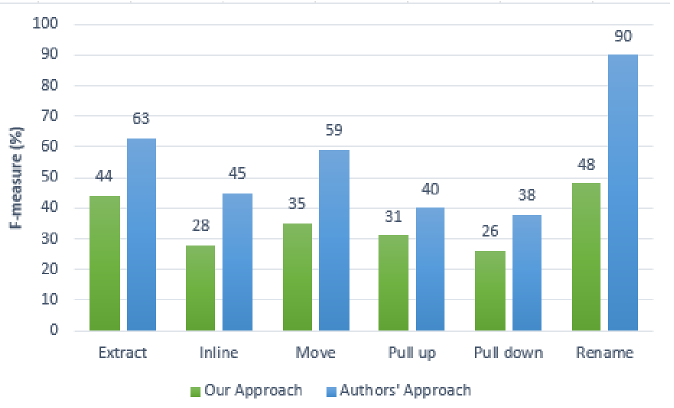


Fig. 7. Visualization of the F-measure for different approaches

and F-measure. For the action of Pull down, our approach had similar precision but lower recall and F-measure. Finally, for the action of Rename, our approach achieved significantly lower precision, recall and F-measure.

It is worth mentioning that our model was designed solely to replicate the authors' approach based on the information provided in their published paper. The variations in the performance of our approach compared to the authors' approach could be due to multiple factors. One such factor was that some of the default parameters employed by the authors were not accessible for scikit-learn's implementation. This was applicable to all three classifiers. Nonetheless, this is not a significant concern as we plan to employ hyperparameter tuning for our optimized solution. Moreover, based on our analysis, it is advisable to consider framing this problem as a multi-label classification problem rather than a multi-class classification problem. Therefore, the current analysis primarily focuses on the reproducibility of the authors' work rather than a complete solution to the problem.

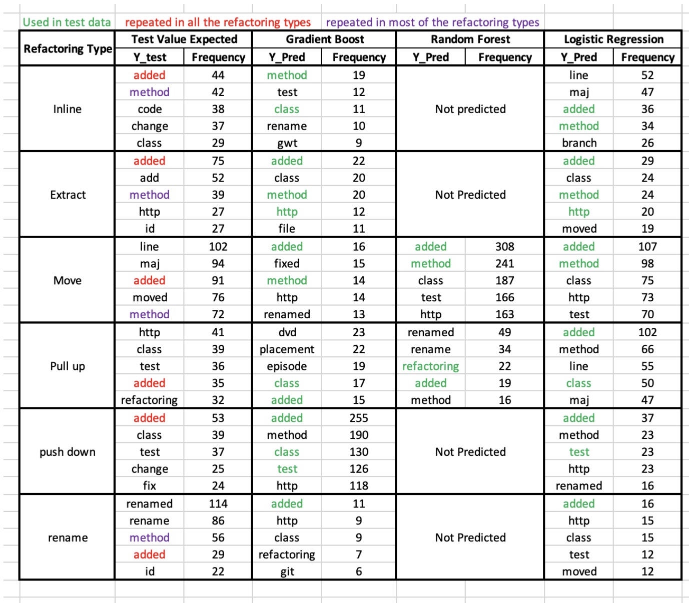
C. RQ3. What are the frequent terms utilized by developers for each refactoring type and how does that compare to our models predicted terms?

In the field of machine learning, the selection of the best classification model for a given dataset is a critical decision. In this study, we compare the performance of three commonly used classification models, namely Gradient Boosted Machine (GBM), Logistic Regression, and Random Forest as they were having the best performance for the authors. In order to determine how relevant, the model's predictions were, we analyzed the frequency of the top 5 most frequently occurring words in the test data for each refactoring type. In the case of the 'Move' refactoring type, a series of steps were taken to analyze the data as follows:

- The first step involved filtering all the records in the X_test dataset for which the output label was 'Move' refactoring in the y_test data.
- Next, the frequency of each word in the X_test data for that corresponding type was calculated, and the top five occurrences were selected.

- The three models were then used to predict the output labels for the X_test data, and the prediction results were extracted.
- Finally, the frequency of each word was again calculated for the X_test data, based on the output which was predicted to be ’Move’ refactoring type.

The comparison between the three models, along with the keyword occurrences in the expected and predicted data, is presented in the table below. These steps were taken to analyze all the refactoring types. The results of this analysis will be useful in understanding the effectiveness of the three models in predicting the various types of refactoring.



Used in test data | repeated in all the refactoring types | repeated in most of the refactoring types

Refactoring Type	Test Value Expected		Gradient Boost		Random Forest		Logistic Regression	
	Y_test	Frequency	Y_Pred	Frequency	Y_Pred	Frequency	Y_Pred	Frequency
Inline	added	44	method	19	Not predicted		line	52
	method	42	test	12			maj	47
	code	38	class	11			added	36
	change	37	rename	10			method	34
	class	29	gwt	9			branch	26
Extract	added	75	added	22	Not Predicted		added	29
	add	52	class	20			class	24
	method	39	method	20			method	24
	http	27	http	12			http	20
	id	27	file	11			moved	19
Move	line	102	added	16	added	308	added	107
	maj	94	fixed	15	method	241	method	98
	added	91	method	14	class	187	class	75
	moved	76	http	14	test	166	http	73
	method	72	renamed	13	http	163	test	70
Pull up	http	41	dvd	23	renamed	49	added	102
	class	39	placement	22	rename	34	method	66
	test	36	episode	19	refactoring	22	line	55
	added	35	class	17	added	19	class	50
	refactoring	32	added	15	method	16	maj	47
push down	added	53	added	255	Not Predicted		added	37
	class	39	method	190			method	23
	test	37	class	130			test	23
	change	25	test	126			http	23
	fix	24	http	118			renamed	16
rename	renamed	114	added	11	Not Predicted		added	16
	rename	86	http	9			http	15
	method	56	class	9			class	15
	added	29	refactoring	7			test	12
	id	22	git	6			moved	12

Fig. 8. Comparative study between the model performance and the keyword frequency

It can be seen from the above table that all the refactoring types have the key words ‘added’ in the top 5 occurrences and the word ‘method’ in four of them. This shows that the keys words used by the developers in the commit messages are not very specific to what type of refactoring was done. Even the training of all the 5 refactoring types would have used these common words even though their TF-IDF scores may be different.

Similarly, the comparison of y_pred and y_test value shows that, there are at most 2 words in each of the refactoring type which it used during the training. For the Gradient Boosted Machine Model, we can see that the frequencies of all the words are high for the push down method. This may imply that the push down method was detected the highest as there is a significant difference.

For the Random Forest the highest frequency was found in Move refactoring type and the next was Pull up method. The remaining types were not even predicted by that model. Similar to the Random Forest model, the Logistic Regression also had 2 refactoring types with significant difference in frequency. One is Move and the other is Pull up refactoring type.

We can see that the ‘Rename’ refactoring type did not even have the word rename being picked up for the prediction of the label. Instead, it can be seen in the pull-down method which may not be very accurate. The results suggest that additional features beyond keywords may need to be included for more accurate refactoring type prediction. Further optimization may improve the performance of the models and provide more accurate predictions for a wider range of refactoring types.

VII. Our Study Design

In this phase, we made some changes to the replicated overall prediction architecture shown in Fig. 1 to perform multi-label classification on the commit messages rather than multi-class classification, which was our proposed solution.

Once the data is ready, the solution design Fig. 9. has four main phases: (1) Data Transformation (2) Text Cleaning and pre-processing, (3) Feature Extraction using N-Gram, (4) Model training and building, and (5) Model Evaluation.

1) **Data Transformation**: Firstly, we have prepared dataset to make it suitable for multi-label classification. We removed the duplicate entries for the same label in the Message.csv using pandas library in Python programming language. Then, columns were added to dataset (dataframe) which were named after each of the refracting types. Corresponding to the commit message, the refactoring methods were binary coded. Figure 10 shows the input samples (x1, x2...xn) and the output samples (y1, y2...y6) which are the 6 refactoring types which has been used.

2) **Text cleaning and pre-processing**:
   - We have converted the case of the commit messages to lower case.
   - To remove the noise, the most frequently occurring words across all refactoring types were included to the stop words list. But, later on they were removed from the list because the model predicted comparatively well with those words than without.
   - Tokenized each word in a sentence using NLTK after the stop word removal.
   - Performed lemmatization on the tokenized data.

3) **Feature extraction using N-Gram**: Performed TF-IDF vectorization on the cleaned, tokenized and lemmitized input data

4) **Model training and building**: We split the data into train and test sets in 80 to 20 ratio and worked on three distinct approaches for multi-label classification are namely, Classifier Chains, Binary Relevance and Label Powerset.
   - Classifier Chains (CC) is a more complex approach that exploits the correlation between the labels by using a chain of binary classifiers. In this method, the classifiers are trained in a chain structure where

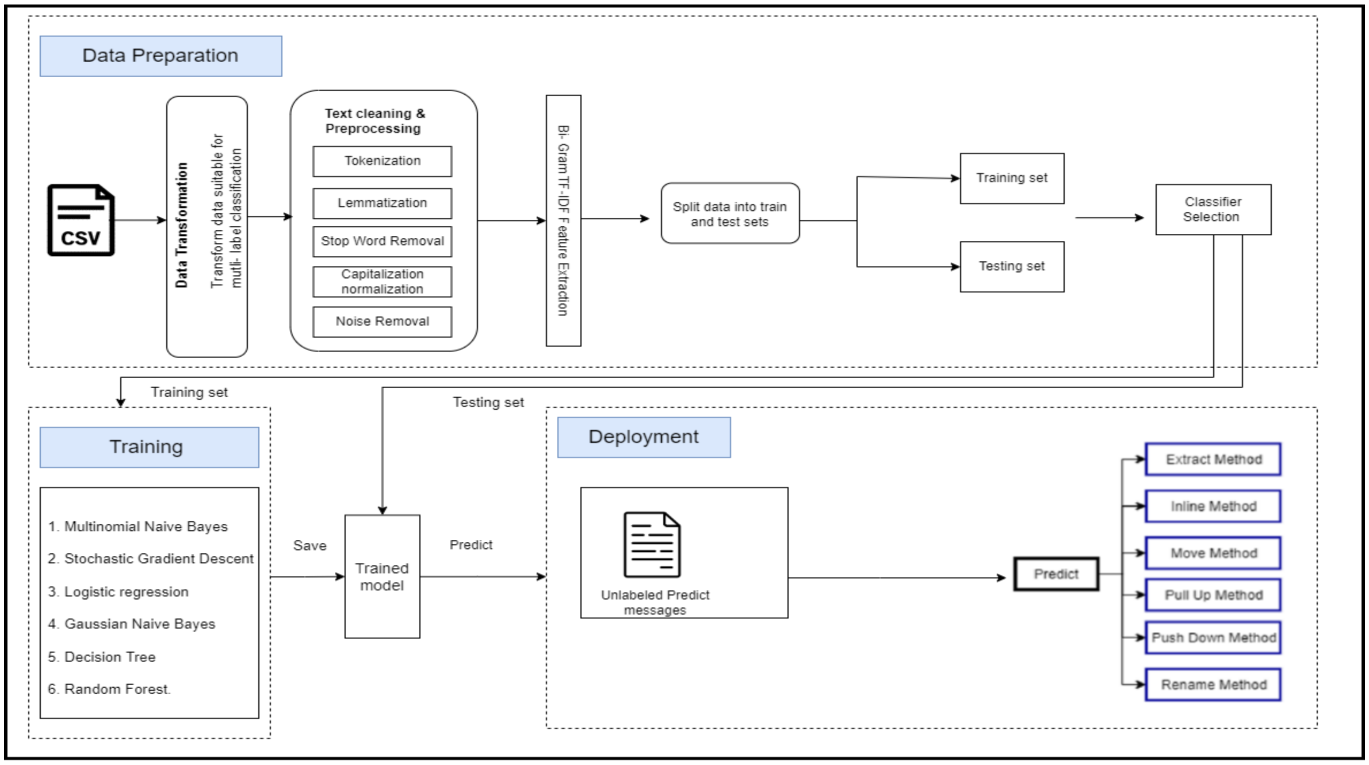
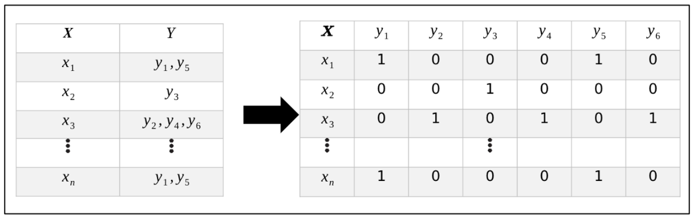


Fig. 9. Overall Study Design



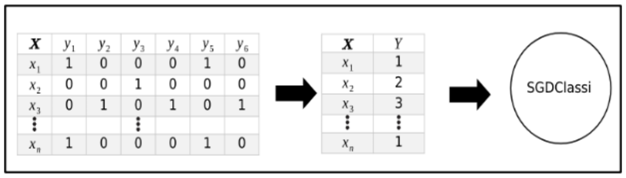
X	Y
x_1	y_1, y_5
x_2	y_3
x_3	y_2, y_4, y_6
⋮	⋮
x_n	y_1, y_5

X	y_1	y_2	y_3	y_4	y_5	y_6
x_1	1	0	0	0	1	0
x_2	0	0	1	0	0	0
x_3	0	1	0	1	0	1
⋮			⋮			
x_n	1	0	0	0	1	0

Fig. 10. Transform data from multi-class to multi-label

the output of one classifier is fed as an input to the next classifier, with each classifier predicting the presence or absence of a single label. The order of the classifiers in the chain is determined based on the correlation between the labels.

- Binary Relevance (BR) is a simple and popular approach that treats each label as a separate binary classification problem. It trains a separate classifier for each label, and the output of each classifier indicates whether or not the instance belongs to that label. The main disadvantage of BR is that it does not consider the correlations between the labels, and may not capture the joint relationships between the labels.
- Label Powerset (LP) is another approach that transforms the multi-label problem into a multi-class problem. It does this by considering each unique combination of labels as a separate class, and then trains a multi-class classifier on this transformed dataset as shown in Fig. 11. Here the dataset for multi-labels will be converted into a multi-class one and then the classification will be performed on it. LP can capture the dependencies between the labels, but it can be computationally expensive when the number of labels is large.



X	y_1	y_2	y_3	y_4	y_5	y_6
x_1	1	0	0	0	1	0
x_2	0	0	1	0	0	0
x_3	0	1	0	1	0	1
⋮			⋮			
x_n	1	0	0	0	1	0

X	Y
x_1	1
x_2	2
x_3	3
⋮	⋮
x_n	1

Fig. 11. Label Powerset Approach with Stochastic gradient descent classifier

Classifiers considered were Multinomial Naive Bayes, Stochastic Gradient Descent, Logistic Regression, Gaussian Naive Bayes, Decision Tree, Random Forest.

5) **Model evaluation**: Due to the fact that multi-label prediction contains the added concept of being partially accurate, we would have different measures to evaluate the algorithms. Accuracy, Hamming Loss, Micro and Macro average values of classification performance were used to examine each classifier in our study. The final well performing model was chosen which was SGD in Label Powerset and the Confusion Matrix was plotted for it as shown in Fig.12

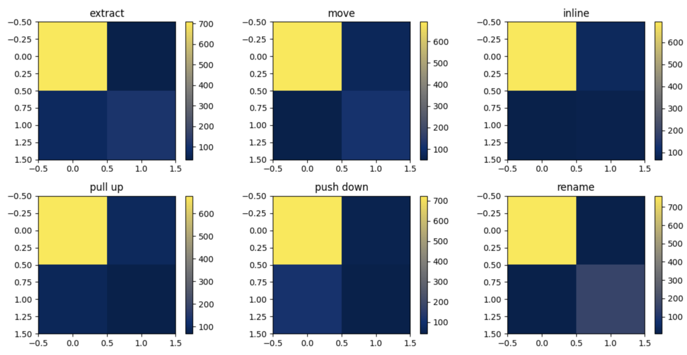


Fig. 12. Confusion matrix for each label classified by SGD model

VIII. Comparative Study

In the preceding sections of this paper, the author's works were replicated through the implementation of multi-class classification algorithms. The results indicated that Logistic Regression demonstrated superior performance when compared to the other algorithms. Subsequently, a more appropriate approach was adopted for the analysis, whereby the concept of multi-label classification was employed, as a single commit message may be attributed to multiple output labels. This section focuses on three different comparative studies, firstly, the performance of multi-class and multi-label classification methods followed by the comparative study between micro-average and macro-average values in multi-label approach and finally the comparison of subset accuracy and the Hamming loss between 3 different methods which are Binary Relevance, Classifier Chains and Label Powerset.

A. Comparative study between Multi-Class and Multi-Label classification

The multi-class classification algorithm showed that the Logistic Regression model had the performance with a maximum accuracy of 35%. The precision, recall and F1 score was as shown below in Fig. 13. for each class. Three distinct approaches, namely Classifier Chains, Binary Relevance, and Label Powerset, were employed in the process of multi-label classification. It was determined that the Label Powerset method, in particular when utilizing the Stochastic Gradient Descent (SGD) model, demonstrated the greatest performance. The accuracy of the method was computed to be approximately 57%. Moreover, the precision, recall, and F-1 score values for each label were assessed individually and are presented in Fig. 14.

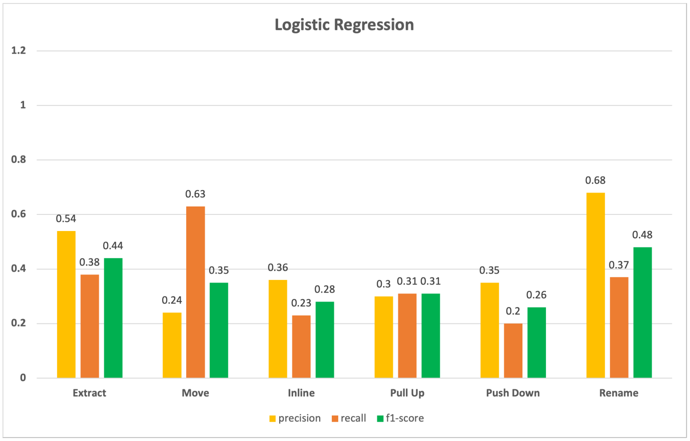


Fig. 13. Classification Metrics report for Logistic Regression Model

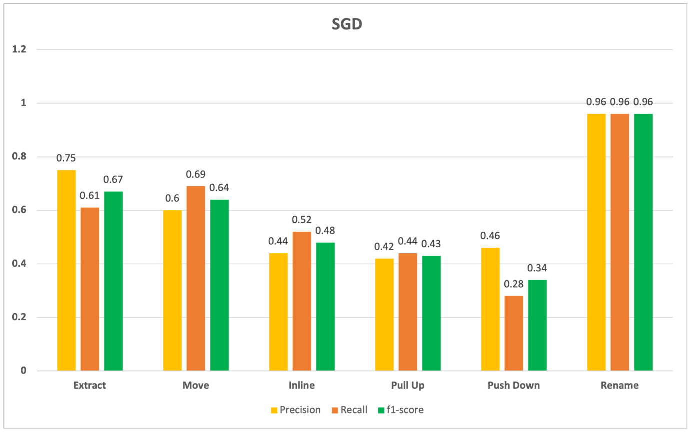


Fig. 14. Classification Metrics report for SGD Model

In both the multi-class and multi-label approaches implemented in this paper, the rename refactoring type exhibited good performance, with high precision and F1-score values. Specifically, in the multi-class approach, the precision and F1-score values for this type were 0.68 and 0.48, respectively, while in the multi-label approach, these values were 0.96. However, the recall value for the rename type in the multi-class approach was only 0.37, which was significantly lower than the recall value of 0.96 recorded for this type in the multi-label approach. Notably, the performance of the rename refactoring type was found to be better in comparison to the author's work because of the usage of words like ”rename” or ”renamed” being used by developers in most of their commit messages.

B. Comparative study on Micro-Avg and Macro-Avg values

One limitation of the comparative study presented in the previous sub-section is that multi-label classification was used as an approach, which makes measuring the precision, recall, and F1-score less appropriate as there can be multiple labels associated with one commit message. However, this limitation was addressed by using the macro-avg and micro-avg metrics for evaluation.

Micro-averaging is an approach that treats all instances and labels equally and calculates the aggregate performance across all instances and labels. In other words, micro-averaging is similar to treating the entire data set as a single binary classification problem and computing the performance measures.

In contrast, macro-averaging calculates the performance measures for each label independently and then averages them across all labels. It computes the precision, recall, and F1-score for each label separately and then takes the average over all labels. This means that each label is given equal weight, regardless of its frequency in the data set.

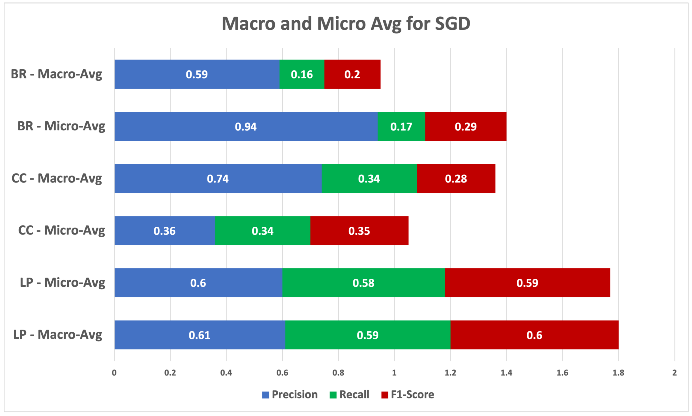


Fig. 15. Micro Average and Macro Average comparison for SGD Model

Figure 15 illustrates the macro-average and micro-average values for the SGD classifier in three different approaches, namely Binary Relevance (BR), Classifier Chains (CC), and Label Powerset (LP). The figure presents the performance of the SGD model, which is found to be equally good across the precision, recall, and F1-score metrics, specifically in the Label Powerset approach. However, in the other two approaches, the performance is noted to be poor in terms of either recall or F1-score.

C. Comparative study between Binary Relevance, Classifier Chains and Label Powerset

As explained already in section 7, Binary Relevance, Classifier Chains and Label Powerset are 3 approaches which are used to work on multi-label classification issues. To determine

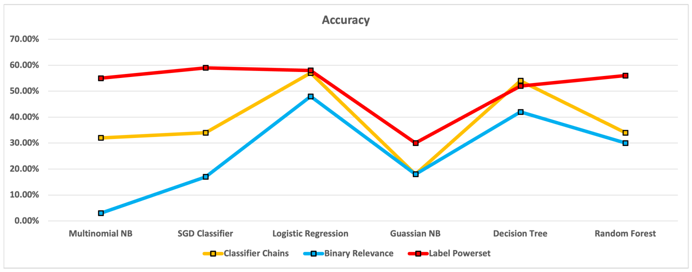


Fig. 16. Comparison on Accuracy

which classification model provided better performance in terms of Hamming Loss Eq.2 and Subset Accuracy Eq.1, a comparison was conducted across all models used.

$$\text{Subset\_Accuracy=} \frac{\text{Count of instances where predicted label matches true label}}{\text{Total number of instances}} \quad (1)$$

$$Hamming\ Loss = \frac{\text{Number of misclassified labels}}{\text{Total number of labels}} \quad (2)$$

Figure 16 displays the results, which indicate that the Label Powerset approach consistently outperformed the other approaches, with the highest accuracy found in the SGD model. Similarly, in terms of Hamming Loss, this approach had comparatively lower values across all models, as depicted in Figure 17.

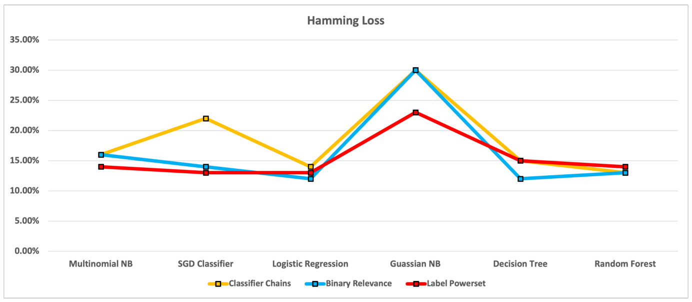


Fig. 17. Comparison on Hamming Loss

In conclusion, this study conducted a comparative analysis of multi-class and multi-label classification approaches for refactoring detection in software engineering. The results indicate that the Label Powerset approach, particularly when used with the SGD model, provides the best performance in terms of accuracy, precision, recall, and F1-score. Additionally, the comparative analysis of micro-avg and macro-avg values highlights the importance of selecting appropriate evaluation metrics for multi-label classification.

IX. Threats to Validity

As with any research methodology, there are potential threats to the validity of our approach. Validity refers to the extent to which a study measures what it is intended to measure and the degree to which the results can be generalized to other settings and populations. In this section, we will discuss several potential threats to the validity of our approach.

In our approach, we aim to use multi-label classification instead of multi-class classification. Multi-label classification is a machine learning technique where an instance can belong to more than one class simultaneously, whereas multi-class classification assigns an instance to only one class. The advantage of multi-label classification is that it can capture the complex relationships between different classes, which can be useful in many real-world applications.

However, the data we have for our analysis has only 4,653 instances with multilabels out of the total 5004, which can be a potential limitation for our approach. This means that our model will have to rely on a relatively small number of instances to learn the complex relationships between multiple labels. This can increase the risk of overfitting and lead to poor generalization of the model to new, unseen instances.

With this we also run into the problem of class imbalance as we modify our dataset to suit multilabel classification. This can also affect the performance of our model, as it may be biased towards the more frequent classes. To address this issue, we can use techniques such as oversampling or undersampling to balance the distribution of instances across classes as future work.

Another area for improvement is in the tuning of our model. Firstly, multi-label classification involves predicting multiple labels for each instance, which can make the search for optimal hyperparameters more complex and time-consuming. This is because the model needs to balance the trade-off between predicting multiple labels accurately and avoiding overfitting or underfitting. Finding the optimal hyperparameters in such a high-dimensional space can be computationally expensive and require a large amount of labeled data. Secondly, the lack of well-established evaluation metrics for multi-label classification can also make it difficult to tune the model. In multi-label classification, there is no clear notion of accuracy or error rate since an instance can belong to more than one class simultaneously. Instead, metrics such as Hamming loss, Jaccard index, or F1-score are often used to evaluate the model's performance. However, these metrics may not be suitable for all applications, and choosing the appropriate metric can be challenging.

Also, the evaluation metrics for multi-label and multi-class classification are different, it may not be appropriate to directly compare the performance of our approach to previous work that has used multi-class classification. This is because the performance of multi-class classifiers is evaluated using metrics such as accuracy, precision, recall, and F1-score, which are different from the metrics used to evaluate multi-label classifiers, such as Hamming loss, Jaccard index, and F1-score with micro or macro averaging.

X. Conclusion

In this paper, we replicated the work of [7] using multi-class classification and obtained an underwhelming accuracy of 35%. To address the limitations of this work and better capture the relationships between refactoring types, we implemented multi-label classification. Our approach utilized three multi-label classification techniques and six supervised machine learning algorithms, to classify commit messages into any number of the six method-level refactoring operations. Our main findings demonstrate that our approach resulted in significantly better performance than the multi-class classification approach. The prediction results for *Rename Method*, *Extract Method*, and *Move Method* were ranging from 67% to 96% in terms of F-measure. However, our model struggled to accurately distinguish between *Inline Method*, *Pull-up Method*, and *Push-down Method*, with F-measures between 34% and 48%. Our findings highlight the need to improve the quality of refactoring commit messages and promote the use of refactoring documentation generation tools.

In the future, we would like to increase the scope of our work to other domains and implement deep learning methods to potentially improve the detection of these refactoring types. One interesting research direction is to investigate incorporating additional features or contextual information into the classification model, such as code metrics, developer experience, or project history. Also, future research could investigate the relationship between detection accuracy and software quality metrics, such as maintainability or reliability.

References

[1] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?" in *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 10–10.

[2] J. Ratzinger, T. Sigmund, and H. C. Gall, "On the relation of refactorings and software defect prediction," in *Proceedings of the 2008 international working conference on Mining software repositories*, 2008, pp. 35–38.

[3] J. Ratzinger, "space-software project assessment in the course of evolution," Ph.D. dissertation, 2007.

[4] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011.

[5] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson, "Comparing approaches to analyze refactoring activity on software repositories," *Journal of Systems and Software*, vol. 86, no. 4, pp. 1006–1022, 2013.

[6] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.

[7] E. A. AlOmar, J. Liu, K. Addo, M. W. Mkaouer, C. Newman, A. Ouni, and Z. Yu, "On the documentation of refactoring types," *Automated Software Engineering*, vol. 29, pp. 1–40, 2022.