Recommendation of Refactoring Techniques to address Self-Admitted Technical Debt

1st Abdullah A Alsaleh
Department of Software Engineering
Data Science Master's Student
Rochester institute of Technology
Rochester, USA
aa6304@rit.edu

2nd Vinayak Sengupta
Department of Software Engineering
Data Science Master's Student
Rochester institute of Technology
Rochester, USA
s4016@rit.edu

3rd Mohamed Wiem Mkaouer
Department of Software Engineering
Project Advisor
Rochester institute of Technology
Rochester, USA
mwmvse@rit.edu

***Abstract*—Self-Admitted Technical Debt (SATD) has become a common and reliable indicator for possible errors in code or codes in need of improvement/optimizations. However, differentiating between SATD comments that reflect the need for refactoring versus those that need handling of other code problems like code smells or anti-patterns is a challenging task due to the similarity in natural language. Along with that, identifying the type of refactoring that needs to be identified from its commit messages will make a considerable improvement for new and current developers of the project in making the right changes, saving time, money and computational resources. Also the domain of SATD comments refactoring has not been researched before. Hence, we address this lack of study with a model that will predict the type of refactoring method that should be applied on those SATD comments that require refactoring to make the process of code refactoring much more efficient. We experimented with multiple machine learning and deep learning models towards tackling the prediction of refactoring labels.We found Random Forest (RF) and Multi-layer perceptron (MLP) to perform the best with a F1 score of 0.73. Hence, showcasing that the model has a good (above average) performance on the prediction of refactoring labels on unseen data.**

***Index Terms*—self-admitted technical debt, code refactoring, recommendation, SATD**

I. Introduction

The goal of all software projects is to deliver high-quality, defect-free software. It is common for software development and maintenance teams to plan their efforts in order to deliver high quality software. For numerous reasons, developers are often pressed into completing tasks before they are ready. Cost reduction, satisfying customers, and competition among others are a few of the reasons mentioned in prior work [18]. Hence, it is not uncommon for developers to leave something untouched only if it is absolutely necessary, so this "Technical Debt" or "hack" results in poor performance and poor maintainability in the long run [18]. In addition to Technical Debt, Potdar and Shihab also introduced the idea of Self-Admitted Technical Debt (SATD) [19], assigned intentionally by developers, but acknowledged in source code comments. Based on their findings, SATDs can be found, depending on the system, in 2.4%-30% of the files. However, it is only 26%-63% of the files that are removed. An instance of this case can be seen in Fig. 1 and 2.

```
  protected void connectIfNecessary() {
-   // can we avoid copy-pasting?
    if (!client.isConnected()) {
```

Fig. 1. SATD comment reporting an issue

```
    body = exchange.getOut().getBody();
+   // TODO: what if exchange.isFailed()?
    if (body != null) {
```

Fig. 2. A Refactoring Method tackling the SATD

A. Technical Debt (TD)

According to the definition provided by Ernst et al. [1] as well as Cunningham [2] Technical Debt can be defined as "not quite right code which we postpone making it right". The reasons for this could be deadline compulsions, inability in producing an optimal solution for a programming issue, or even lack of required resources. Technical Debt plays a sensitive role in software quality because of its massive impact on the development life cycle when adding some new features or fixing a problem. Also, it will affect the experience of the main factor of the software which is the end-user. These problems in the code need to be fixed, and always dealt with when exposed for a healthy software quality.

B. Self Admitted Technical Debt

Technical Debt (TD) becomes a Self Admitted Technical Debt (SATD) when the developer mentions or "admit" that there exists a non-optimal code in the system near the TD [3]. SATD may not be immediate errors in the code, but they have the potential of causing major errors and issues in the future addition of features and even debugging [4]. SATD, due to the implementation of a "make-do" code can also confuse when a new developer starts working on the same project affecting productivity. The process of implementing these changes includes locating the source code that should

be modified and applying the optimum changes on the located source code is known as Refactoring.

C. *Code refactoring*

Refactoring is ”the art of improving the design of a system without altering its external behaviour” [5]. There have been several studies that have concentrated on the practices that are followed by developers while refactoring their code [6] as well as on the documentation for these practices [5]. Such studies leverage enhanced insights into concepts like code improvement and even possibly make it more impermeable to errors. Besides, some existing studies [1] [7] of self-admitted technical debt SATD have widely focused on the type of the changes of method that occurs in source code without looking to the refactoring that may be achieved by the developers itself. Unfortunately, there does not seem to be present a fundamental form of classification for software refactoring types from the self-admitted technical debt by the developers. Also there seems to be studies missing where SATD based comments are classified before to evaluate if they require refactoring or not. And also to predict a recommended refactoring method towards those comments that indicate an area of code where refactoring is actually required.

Our system is be able to recommend refactoring methods that can help solve the intended SATD. Our system has implemented library of 10 most frequently utilised refactoring methodologies across various software development fields. Some of the more frequent methods used are - Change Attribute Value, attributes defined within XML tags can be modified with this method [1], Move Class, this refactoring leverages a ’Copy’ function that creates a copy of a class, file, or directory in a different package. You can also copy files, directories, or packages in new directories or packages [2], Rename Method, this function allows us to rename identifiers for source code elements like fields, local variables, methods and types, when we would like to safely rename without having to find all its related instances [3].

D. *Recommendation of Refactoring Techniques to address Self-Admitted Technical Debt*

This project introduces a system that given an SATD comment, will predict refactoring method labels that will help in tackling that issue. This will make future developments and changes needed to be made, efficient and relevant. We see this to be particularly helpful in projects and organizations with big developer teams and code-bases that are difficult to maintain or make changes in. The project leverages machine learning/deep learning model architectures and training the model with a valid data set for this purpose. For the immediate experimental studies, we conducted a series of experimental studies using first machine learning architectures like Random Forest, Logistic Regression, Multinomial Naive Bayes and Support Vector Machine. Based on the results of these models, we then even dove into advanced deep learning architectures of Convolutional Neural Networks and Long Short Term Memory models for better performance and accuracy. This form study will support developers in this role with automatic suggestion tools to identify if the refactoring issue or not, this tool helps to reach the best quality of the software development process. The rest part of the paper is organized as follows. The solution approach and the architectures are briefed in Sect. 4 and Sect. 6, Sect. 5 describes the implementation. Sect. 9 analyses the results and main findings of the study.

[1] https://www.jetbrains.com/help/webstorm/change-attribute-value.html
[2] https://www.jetbrains.com/help/idea/move-refactorings.html
[3] https://docs.microsoft.com/en-us/visualstudio/ide/reference/rename?view=vs-2022

II. Related Work

Code refactoring today is the one practice that determines the longevity of a software today and hence, that of an organization selling that as a product. Since this very practice is so versatile and transferable, it can be categorized under multiple major software engineering fields. The current studies that explore the applications of code refactoring can be categorized into five groups based on its characteristics as the following:

A. *Code Clones*

Code clones have become an obstacle when it comes to source code maintenance. One of the main reasons being they co-evolve, change to one does not necessarily make it to the second hence, creating major inconsistencies and conflicts across the code causing errors. It is a non-optimal form of code reuse and often leads to Self-Admitted Technical Debt (SATD) as well. Hence, code clones should always be refactored to address better software development practices.

*1) **Clone Refactoring**:* Authors Mondal et al. [8] observed a lack of research on how clone refactoring can affect system performance and discussed how code cloning is used and mentioned that developing the supported languages of the clone refactoring tools as increasing the code base.

We noticed that the authors have concentrated only on code clone for refactoring, while we are not restricted to any one instance of the refactorings. The paper also claims that automated code refactoring is problematic, while we offer an effective solution against that very issue. Also according to authors [8] code clones are considered to be a non-optimal form of refactoring, and leads to SATDs.

B. *Source Code Refactoring*

Source code refactoring is the simple task of changing the structure of the code without changing its functionality. Many methods like Rename Method, Extract Class, and Push down Method exist in aiding this very crucial principle practice. This very developmental exercise benefits in making the software easy to understand, locating bugs, and improving the overall consistency of the codebase.

*1) **Refactoring and Code Change Relationship**:* Authors Bavota et al. [9] discovered a gap where the studies only took the inputs of software developers and analyzed the software repositories themselves. Hence, they proposed a quantitative

investigative approach to study the relationship between different categories of code refactoring and code change orders. Utilizing three software projects: (1) APACHE ANT, (2) ArgoUML, and (3) APACHE XERCES-J3. They found bug fixing results in code comprehensibility and sustaining increases, and the success of new code implementation by source code improvement to coherence and Object-Oriented Programming adherence.

However, the paper grossly underestimates TD as a threat, while ours prioritises SATDs. Also, the authors have concentrated on reasons behind the refactoring rather than the practises themselves.

C. *Software Refactoring Recommendations*

Code refactoring is a very healthy practice to maintain code quality and reduce complexity. Unfortunately when dealing with humongous code bases in software, physically searching for code smells or locations needed for refactoring becomes infeasible as it is always not possible to know the insides of the software at all times. Hence, having a tool or an additional feature that helps not only in parsing the code base and detecting the required refactoring locations but also based on syntax and the nature recommending the required type of change to be implemented, will help the developers in increasing code maintenance efficiency and also reduce error-prone changes.

*1) **Feature Request based Code Smell Refactoring**:* An approach is proposed by Nyamawe et al. [10] to reduce code smells in the source code by using a refactoring tool and helps developers to recognize the previous refactoring. This approach consists of two classification tasks: a binary classification, and multi-class classification to predict the refactoring labels. Later, many approaches were proposed to compare the refactoring solutions by reducing their influence on traceability. These studies claimed that many cases operate more reliably on the proposed approach than the traditional one that uses the code metrics-based.

The work of this study is what inspired ours as well. The approach of the paper returns a refactoring label in reference to the entire source code scope. We achieve the same but more specifically tackling the SATD domain.

*2) **Code Smell Label based refactoring**:* Another study by Authors Nyamawe et al. [11] discussed different approaches to compare the refactoring solutions by reducing their impact on traceability. Their solution architecture had suggestions for several refactoring methods for an individual code smell. Then based on traceability it measures the impact of the suggested solutions virtually on source code and traceability which share suitable recommendations based on the code metrics and the enhancement of entropy. Their proposed approach is reasonable since it makes the workflow easier and the process becomes fully automated. On the other side, 25 out of the 85 code smells preferred the baseline approach versus the proposed approach and this is mainly due to not all developers making their selection decision based on the traceability and code metrics and refactoring because traceability is not an accurate factor to measure traceability.

The author's work tests code refactoring without checking if it is even needed, while our solution will check for a refactoring requirement first. Recommendations are made on source code traceability while we make them on the SATD requirements hence, they are more applicable and developer friendly

D. *Code Refactoring Tools*

Refactoring of code by developers themselves is reliable but inefficient for the long run with further increase in the software complexity and size. Hence, in those cases, it is easier and more reliable if refactoring is implemented using robust tools that can automate the desired changes and also locate these change areas faster. Tools can be in many forms - independent software, IDE plug-ins, and even recently, automated bots.

*1) **Code Refactoring Detection Software**:* Authors Krasniqi, R., and Cleland-Huang, J [13] performed a new code refactoring tool "CMMiner" which recognizes the code commit messages. CMMiner detected some refactoring types which were not discovered by RMiner like Remove Parameter, Hide Method, and other types.

The paper seems to concentrate only on detecting code refactorings through tools. We recommend the refactoring labels based on the SATD comments. They also focused on bug fix comments while we are concentrating on SATD comments which is probably a larger scope of code.

*2) **Bot Automation for Code Refactoring**:* In The work of Marvin Wyrick and Justus Bogner [17], they developed an autonomous bot for source code refactoring that catches the clear restrictions of previous refactoring bots. The authors evaluate the bot's success by three sets of success factors: 1) Trust, familiarity, and non-patronizing 2) Perceived utility, and 3) Manual efforts for developers.

E. *Micro-services Architecture*

Micro-services Architecture is an approach that includes building systems from smaller services and communicating over a protocol. Every micro-service here represents a unique function. The most challenging of decomposing(migrating) current software to this form of architecture is the very refactoring of the data structures and code base itself. It is also very hard to figure out as to which approach one must take in the decomposition as it varies case to case and also size.

*1) **Micro-services Architecture based Refactoring**:* Some studies were proposed for understanding which micro-service architecture refactoring approaches to use it beginning with reviewing all the current Micro-services-related refactoring techniques In a work by Brereton [16], has proposed a three-step reviewing process that has been imbibed by the authors: planning, conducting, and documentation. Though the study design is noteworthy it has several limitations. Like The review, the authors conducted on the previous literature turned away from the set guidelines of systematically related works

reviewing. Another issue in the design guide is that the solution has only theoretical standing.

The study however is slightly lacking as it is just an empirical study that concentrates on micro-services refactoring, while ours is an experimentation surrounding SATD refactoring.

By conducting a thorough literature survey, we learned that the main consensus towards modern code refactoring practices is aimed at generic software code smells or defects eradication and code maintenance. The studies concentrated more on solving issues on making the code refactor identification and the refactoring method automated and more accurate, but again in the domain of the source code base as a whole only. While the scope of the previous studies is well informed and noble, unfortunately,none of the work dived deep into the more subtle, human,and often procrastinated corner of the code base, which is Self Admitted Technical Debt (SATD).

III. Approach

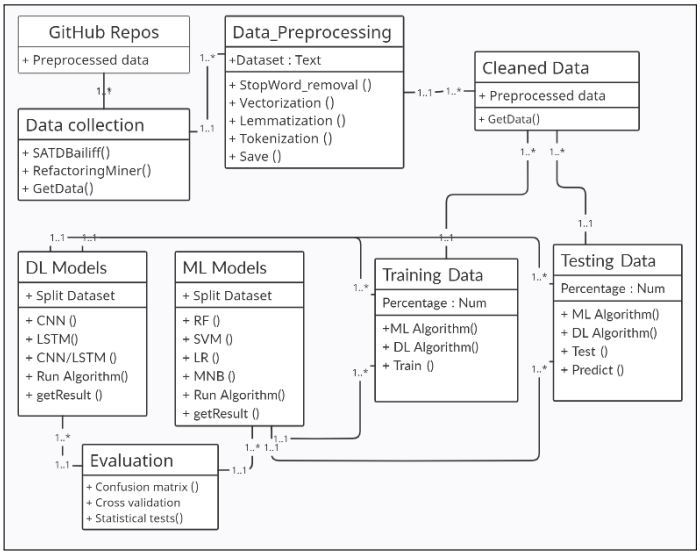

Fig. 3. Domain Model Diagram for Recommendation of Refactoring Techniques to address Self-Admitted Technical Debt

A. Data Collection

Like any successful project in software engineering we start with the data collection, this begins with selecting a number of open-source Java projects. The selected repositories are then fed into a tool named SATDBailiff [4], provided by our faculty advisor Prof. Mohammad Mkaouer. The tool will automatically mine out Self-Admitted Technical Debt SATD commit comments and related information and compile it into a CSV file for use. The tool utilizes JGit, a full java implementation of the git version control to pull out the java source files from the given project repository. After getting the files, SATDBAiliff then uses a Java parser for skimming through the files and mining out all the source comments from them. Finally, a custom-built SATD analyzer identifies the SATD comments from the rest of them. It should be noted all these 3 steps are automated and implemented within the tool. The tool is executed by its JAR file using a command . This method is efficient as it helps cut down the time towards physically parsing through each project commit and then categorize the SATD ones from the others which again is prone to interpretation errors. Hence, using this tool that utilizes Natural Language Processing at its core, is very reliable and saves time and effort.Since we are focusing on Java projects, we have utilized another open-source tool called Refactoring Miner [23] that classifies and detects the different refactoring types in the commit history of different Java projects. Refactoring Miner tool takes the list of commits which we extracted using SATDBAiliff as an input, and Refactoring Miner returns a list of refactoring types applied between provided commits.

[4]https://github.com/smilevo/SATDBailiff/

B. Data Preparation

Having obtained the raw dataset, we then converted it into a more understandable and workable format. We started by conducting extensive Exploratory Data Analysis to understand the data completely so that we can apply the applicable preprocessing techniques for it. Next, we applied various Natural Language Processing techniques of Lemmatization, Vectorization, Stop Word Removals, Data Sampling like MLSMOTE and even simplifying the data complexity through Binarization of the column values. These techniques created a more cleaned data free from elements that would waste resources and also created a more workable format of the data for the model to utilize and interpret and learn from the data accurately.

C. Refactoring Types Label

The principle task our model achieved is - given there is a need for refactoring, it returns the type of refactoring label for that particular SATD to be resolved. This recommendation is achieved with the help of a machine learning or deep learning architecture. The right model has been obtained through a comprehensive experimental study of a number of models from the two architectures. We utilized the models of Logistic Regression, Support Vector Machine, Multinomial Naive Bayesian Network and Random Forest from the machine learning architecture. Also, we looked at advanced deep learning models of Convolutional Neural Networks and Long Short Term Memory.

D. Approach Validity

The above-mentioned approach is towards a new solution in the domain of tackling Self Admitted Technical Debt. There has been numerous research in the field of tackling code defects in a generic view of the code scope in the form of code refactoring, but there is no research focused on SATD commented code scopes. Hence, our solution which is inspired by the past state-of-the-art approaches and applying them for SATD seems to be a promising and sensible endeavor. The reason we feel that our methodology for code refactoring recommendation is safe for approach validity because as compared to past works is that past researches have concentrated more on the theoretical side of SATD, involving numerous

empirical studies regarding the identification and possible solutions to it. But leveraging statistical or predictive models like machine learning have not been explored, which makes our endeavour novel in its nature. The major challenge for the research is that the data is entirely in natural language making refactoring-need classification difficult and also like generic NLP projects we cannot do away with the technical jargon and interrelations to the source code as they are the information that will help map out the area of code that needs to be understood. Also since, for our first phase of project experiments we are studying with five different classification models, hence, we feel it is a broad and heterogeneous set of experiments and is a valid approach.

IV. Implementation

A. Data Collection

For the raw data collection, We utilized an open-source tool named SATDBAiliff, provided to us by our project advisor Prof. Mohamed Wiem Mkaouer. The tool is specifically designed towards automatically parsing through the projects of a given project repository and mine out SATD classified source code comments. We are restricted towards Java projects specifically owing to the fact that SATDBailiff is designed specifically for that language for. The tool utilizes JGit, to pull out the java source files from the given project repository. After getting the files, a parser is used for mining out all the source comments from them. Finally, an SATD analyzer identifies the SATD comments from the rest of them. The resulting output of this tool gave us a CSV file with the source code methods, the SATD comments relating to those methods, and also if and how those SATD comments (if any) were resolved.Next , we utilized Refactoring Miner [23] to find refactoring types in the list of the project commits. Refactoring Miner iterates through a repository's commit history in order to find refactoring types by comparing the changes made to Java source code files.

B. Data Storage

A MySQL database system named SATDData has been created for the data storage. The reason for this was primarily because the output raw data by SATDBailiff was to be received as an SQL query into our local database. Another reason we did this because initially we utilised it only to receive the raw data, but now since we applied a wide range of data engineering techniques, and experimenting with them, we maintained a record of those versions of the pre-processed data for understanding the differences when we then compared them to our future works as well.

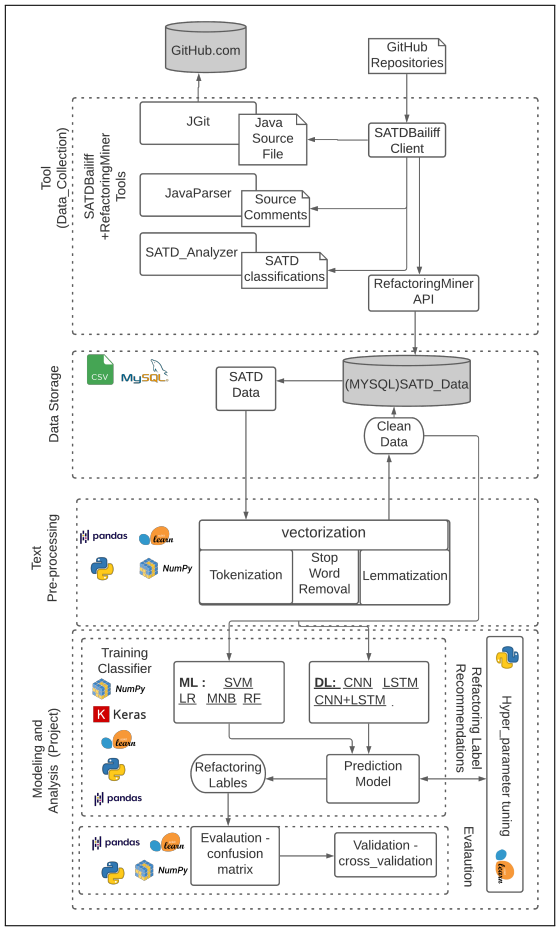

Fig. 4. Architecture Diagram for Recommendation of Refactoring Techniques to address Self-Admitted Technical Debt

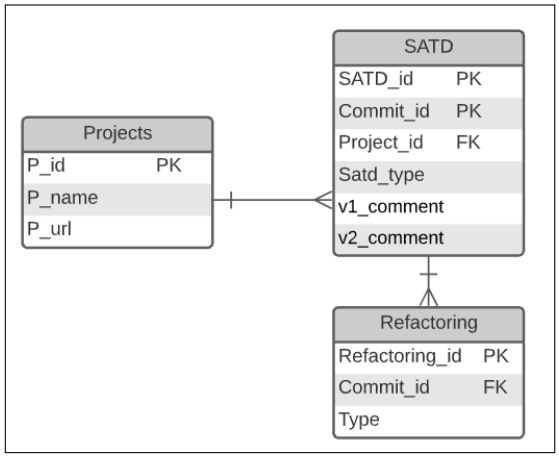

Fig. 5. MySQL database table structure

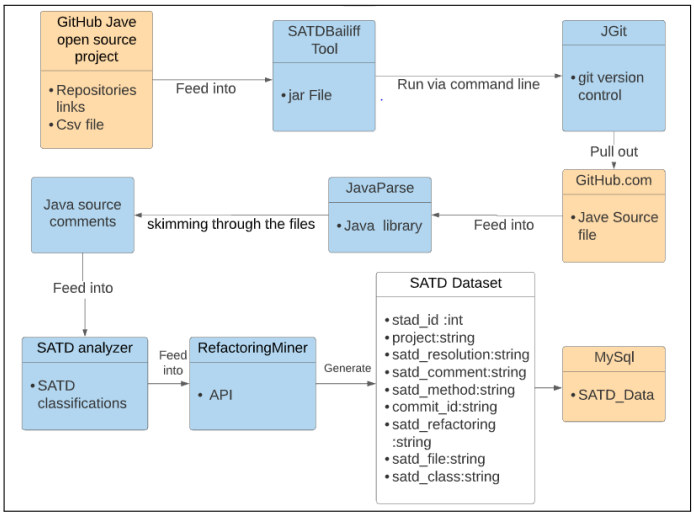

Fig. 6. Ingestion layer for Recommendation of Refactoring Techniques to address Self-Admitted Technical Debt

C. *Text Pre-processing*

Having collected the raw data and also setting up the data storage system, we are now ready to conduct analytical research, modification and manipulation of the data towards our final statistical modelling around it. For this, the following pre-processing methodologies were applied in our first phase of experiments:

*1) **Exploratory Data Analysis (EDA)**:* EDA is one of the most important steps in any data related research as, it is the collective set of actions that will help us understand the contents of the data, relationship between different variables across the data and how these values and relationships can help create the models and solutions. For our case we asked a few research questions that we tried answering through an extensive EDA:

What are the new unique refactoring labels? We are creating a recommendation system which can recommend refactoring from a set of 10 most frequently utilised refactoring methods across several open-source software engineering projects. These are: Rename Parameter, Change Return Type, Change Variable Type, Rename Attribute, Rename Class, Rename Variable, Change Attribute Type, Change Parameter Type, Rename Method, Move Class.

What are the most utilized refactoring methods? As we can see from the fig 7, that Change Attribute Type is the most frequently used/occurring refactoring method across all others. It is also interesting to notice that the refactoring methods increase in an almost linear fashion in strength. This also indicates a gross imbalanced data problem that will need addressing.

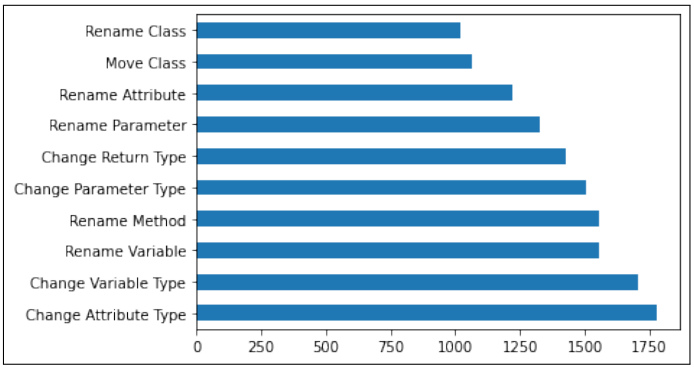

Fig. 7. Refactoring types and their frequencies

Utilising the Refactoring methods, what was the most common end-results? Out of the total 14155 instances of different refactoring method applications, 'SATD_REMOVED' or SATD getting successfully eliminated was the most form of end resolution, which is a good sign.

It is followed by the 'SATD_CHANGED" resolution which entails instances of SATD which have changed in their nature, meaning that the refactoring method helped solve the original problem but may have caused another in its place. This could be a common result of many internal relations being affected by the refactoring.

The next resolution is 'CLASS_OR_METHOD_CHANGED' which showcases what was the end result of the refactoring methods themselves, and these instances most commonly ended with the entire class or method getting changed/refactored. And lastly, we only have 2 instances where application of the refactoring methods yielded no results or changes in the 'None' resolution.

*2) **Data Cleaning**:* We initiated our cleaning process by the removal of stop words and special characters from the SATD comments. This simply seeks to eliminate words that have a substantial impact on the phrase, as well as unnecessary words. Typically, the stop words are the articles and pronouns. We also removed rows with empty values for the columns of resolution or refactoring type.

*3) **Tokenization**:* Tokenization basically means breaking up a sentence, paragraph, or text document into smaller pieces, such as individual words. A token is the individual part (split from the document) of the larger entity. A token can be a word, a number, or a punctuation mark. When tokenization is performed, word boundaries are located to form smaller units. Word endings and word beginnings represent the end of a word and the starting point of the next. As a first step in text pre-processing, these tokens are considered as a one-way means of stemming and lemmatizing.

Tokenization set the stage for further pre-processing for lemmatization and vectorization, as we utilised a self-written split function to separate the sentences from the SATD comments, separating the important TD comment format, eg: FIXME, TODO.

*4) **Lemmatization**:* Lemmatization is an important task in NLP research field, which is the process of mapping multiple versions of the same word to its root, which is called a lemma. The WordNetLemmatization approach is used in this research.This method's goal is to find the structural semantics link between the words in the data set. This was important to us, as SATD comments are really subjective to developers, hence, we had to map words of similar phonetic or lexical style together. Example: for an SATD comment if words like fixme, FIXME occur anywhere in the comment, they are mapped to the FIXME category of TD comments. This is done for all present TD categories in the 'v1_comment' and 'v2_comment' columns.

*5) **Vectorization**:* We made use of the TF-IDF architecture for vectorization of the column section of the data, since for each jargon word (code syntax) we achieve a weight importance value and that will help with highlighting certain syntax words or indicative words that will help with refactoring label prediction. We applied N-grams for the model, so that we can filter down the search area, making it more efficient.

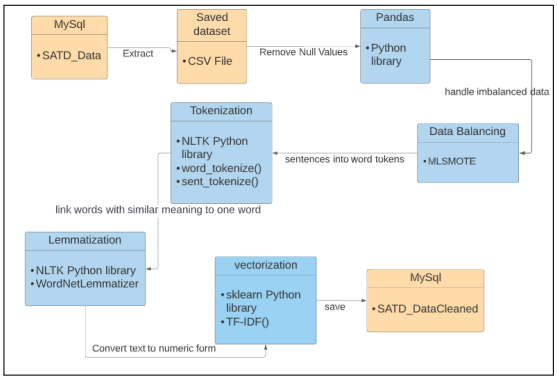

Fig. 8. Data cleaning layer for Recommendation of Refactoring Techniques to address Self-Admitted Technical Debt

*6) **Data Imbalance**:* Class imbalance in the data set can cause uneven training of the classes and lead to low performance results in different metrics such as F1 score, Precision and Recall. The figure 7 shows the frequency of each class in data set.

MLSMOTE

Multi-label Synthetic Minority Over-sampling Technique (MLSMOTE) is one of the most widely used and successful data augmentation methods in the case of multi-label classification. Since we are dealing with a multi-label classification in this project we have utilized the MLSMOTE [20] Techniques to handle the data imbalance issue.

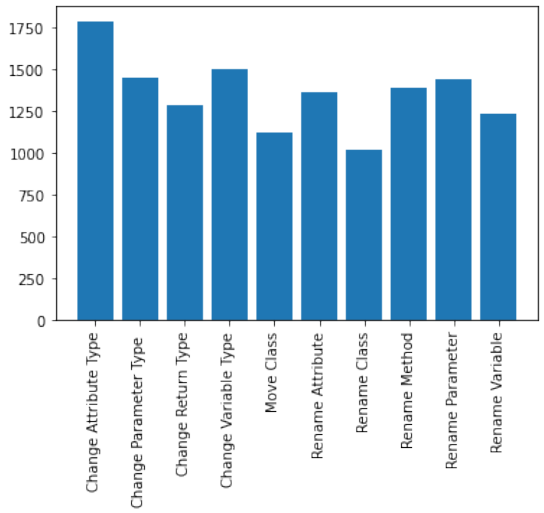

Fig. 9. Classes distribution After Applying MLSMOTE

D. *Modelling and Analysis*

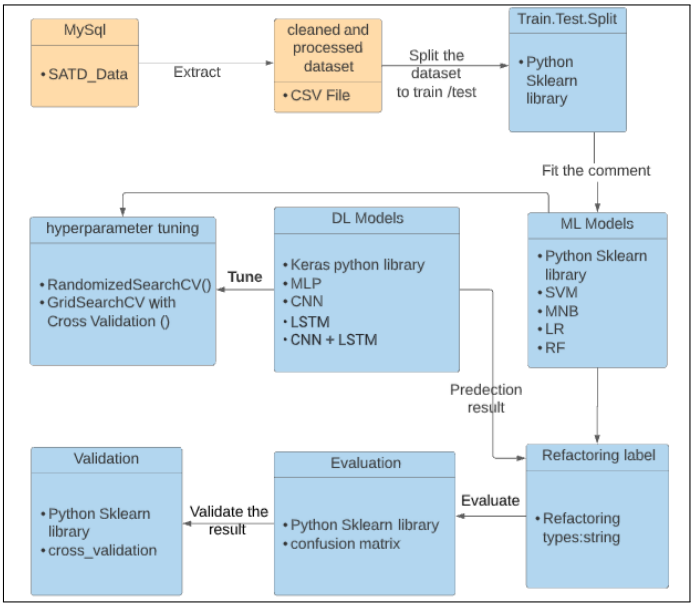

Fig. 10. Modeling and Analysis layer for Recommendation of Refactoring Techniques to address Self-Admitted Technical Debt

The Modelling and Analysis phase is the main experimental scope of our solution architecture, here we will be utilizing models machine learning (ML) as well as Deep Learning (DL) techniques as for our first phase of experiments towards our project. This phase is in accordance to the Refactoring Label Recommendation for given SATD comments.

E. *Refactoring Label Recommendation*

We will recommend the refactoring type label that will most suitably solve that particular SATD issue. The following models will be experimented with:

*1) **Support Vector Machine**:* Since we are converting our paragraphs into word vectors of unique numerical values, we will utilize SVM as it is specifically designed towards the classification of vector space values, and also because SVM has been useful when it comes to text classification across literature's like this one [14]. Since SVM is primarily a binary classifier, we will be converting our preprocessed data set by using dummy variables for the unique classes of labels. We have utilized the SVM algorithm from the scikit-learn [21] library on the encoded text in order to assign it to a specific class. The parameters that we used are the default parameters, which are listed in the Table I.

*2) **Multinomial Naive Bayes**:* Because of its fast learning rate and simple architecture, Multinomial Naive Bayes has been widely used in NLP tasks when compared to other Machine Learning algorithms.In text classification issues, the MNB assumes that each word in a phrase is independent of the others. This indicates that we are no longer considering full phrases, but rather individual words. Our inspiration for using MNB comes from this study [15]. Where it provided with good results towards multi-class classifications in text with a class imbalance problem. Similar to our SVM model we will also be producing dummy variables to take care of the classification model restrictions.We have utilized the MNB algorithm from the scikit-learn library [21], the parameters that we used are the default parameters, which are listed in Table I.

*3) **Logistic Regression**:* Another classification approach used to tackle binary classification problems is logistic regression , We felt that a LR model would be a great base model for other models to be compared to. Logistic Regression utilizes a sigmoid function to process the input features through a weighted combination.The sigmoid function any real nymber to number between 0 and 1 . We have Implemented the LR algorithm from the scikit-learn library [21], the parameters that we used are listed in Table I.

*4) **Random Forest**:* Since RF is well capable of multiclass classification, we felt it would be a good base model along with SVM to compare the performances of all the models. One big advantage of using the random forest is that while growing the trees, the random forest adds more randomness to the model. When splitting a node, The RF will look for the best feature from a random subset of features rather than the most important feature which will lead to a better model.We have utilized the RF algorithm from the scikit-learn library [21], the parameters that we used are listed in Table I.

It should be noted that with the use of multiple models, we are covering a heterogeneous scope of experiments.

*5) **Convolutional Neural Network**:* Convolutional neural network (CNN), which is a type of artificial neural system that has enhanced to be powerful in several computer vision tasks, is grabbing attention beyond a diversity of fields, including NLP. CNN is composed to be adaptive and automatically acquire spatial hierarchies of characteristics within back-propagation by utilizing various building blocks, like pooling layers, convolution layers, and completely connected layers.In case of text classification without neural nets, it takes each word and count how many times it occurs in each document, this means it transforms the string into a vector where the length is the number of words, and it causes a loss of the words' order. The other transformation way is to encode every character in the text, which is a great method, however, it causes multiple issues due to a lot of spaces in languages divided into a set of numbers and in this case, it transforms the sentence words into different numbers based on the word count in the sentence, so, the word will be transformed into the same number always . [22] RNNs are great at predicting and forecasting what is the following to happen in a sequence while CNNs can acquire to analyze a sentence or a paragraph [22]. Although there is complex text in the dataset when we are attempting to predict the refactoring type, the CNN model may be adequate and even more reliable regarding the computation. We have utilized the CNN algorithm from the Keras library, the parameters that we used are listed in Table II.

*6) **Long Short-Term Memory**:* Long short-term memory (LSTM) is a recurrent artificial neural network architecture employed in the section of deep learning. LSTM networks are well suited for organizing, analyzing, processing, and producing predictions regarding time series data due to the potential for unknown delays between significant events in a time series.Long short-term memory (LSTM) has lately become a successful tool among NLP researchers for their superior knowledge to model and learn from subsequent data. These numbers showed the most recent results on various general criteria ranging from sentence classification and various tagging problems to language modelling and sequential predictions. LSTM proposes to resolve the RNN obstacle called gradient vanishing and bursting. In text classification, the good adherence to memorizing certain styles of LSTM performs somewhat better. As with each additional NN, the LSTM can have various unknown layers and as it moves within every layer, the appropriate data is stored and all the unnecessary data gets dismissed in every individual cell.We have utilized the LSTM algorithm from the Keras library , the parameters that we used are listed in Table III.
For the implementation see: GitHub

Models	Parameters
SVM	kernel='rbf' class_weight= 'balanced' c=10
RF	n_estimators=100 criterion ='gini'
LR	penalty='L2' C=1.0
MNB	alpha=1.0 fit_priorbool= True
MLP	activation='relu' hidden_layer_sizes=(100,) max$_iter$ = 300

TABLE I
CLASSIFICATION MODELS AND THEIR PARAMETERS

Layers	Output Shape	Kernel Size
dense	5000	-
dropout	5000	-
$dense_1$	1000	-
$dropout_1$	1000	-
Conv1D	486 × 1000	32
GlobalMaxPooling1D	1000	32
dense_2	600	-
dropout_2	600	-
dense_3	300	-
dropout_3	300	-
dense_4	100	-
dropout_4	100	-
dense_5	60	-
dropout_5	60	-
dense_6	10	-
dropout_6	10	-

TABLE II
CNN ARCHITECTURE

Layers	Output Shape	Kernel Size
Embedding	1000	32
spatial_dropout_1d	1000	32
dense	1000 × 5000	-
dropout	1000 × 5000	-
lstm	196	-
dense_1	600	-
dropout_1	600	-
dense_2	100	-
dropout_2	100	-
dense_3	10	-

TABLE III
LSTM ARCHITECTURE

V. DATA SETS

The data has been collected by the combined effort of 2 open-source tools, SATDBailiff and RefactoringMiner. The SATD-Bailiff program detects SATDs from method comments based on a machine learning model, then tracks the SATDs' span (from their occurrence to resolution). RefactoringMiner [24], [25], is a Java library/API that detects refactorings applied to a Java project.

The main columns of the data are the 'resolution', 'v1_comment', 'v2_comment' and 'refactoring_type' columns containing the end result of the refactoring, the comment before refactoring, the comment after refactoring and the refactoring method itself, respectively. We will be working with 10 unique refactoring label classes. The data set of 14156 rows with a unique instance of comments for each corresponding label.

VI. RESULTS

With the data set processed and binarized, we tried to break the problem down to a simpler problem-solving method i.e. binary classifications for each refactoring method. we had to convert our data set as well to be able to work with the solution models. Hence, we created dummy variables out of each of our extraction labels. This will help us have binary numeric data, that indicates the refactoring correspondence to SATD comment. This methodology helps to reduce the complexity of handling the textual data and also allows for a larger number of machine learning models to be experimented with. To investigate which classifier will be effective in predicting and recommend the refactoring type label that will most suitably solve the SATD issue. We evaluated the performance of the four machine learning and three deep learning models as mentioned before, Support vector machine (SVM), Logistic regression (LR), Random Forest (RF), multinomial naive Bayes (MNB), multi-layer perceptron (MLP), convolutional neural network (CNN) and long short term memory (LSTM). The following table shows the performance of each classifier used in the experiment in term of F measure, and accuracy

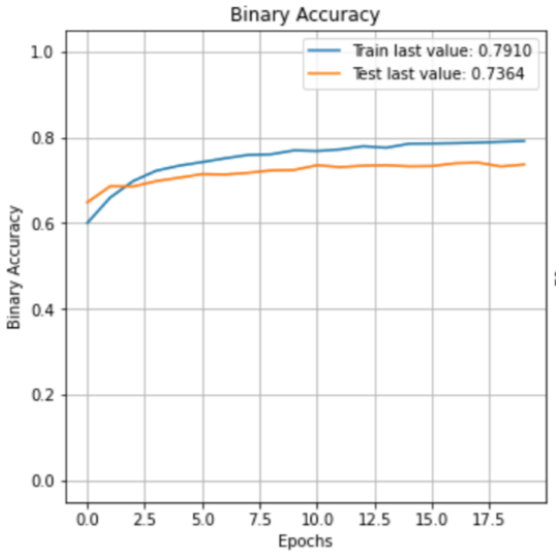

Fig. 11. Training and validation accuracy over epochs :CNN

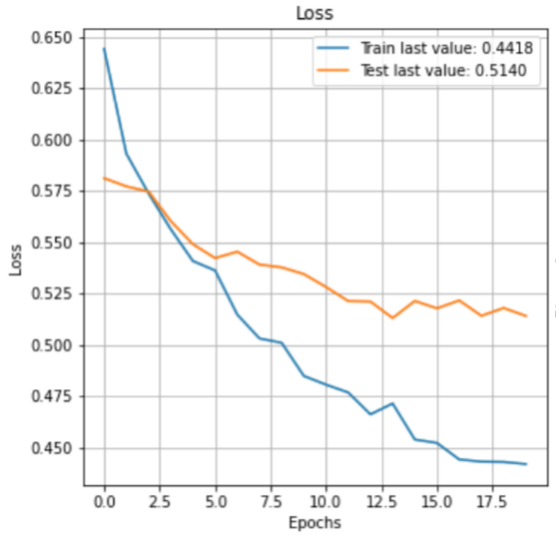

Fig. 12. Training and validation loss over epochs :CNN

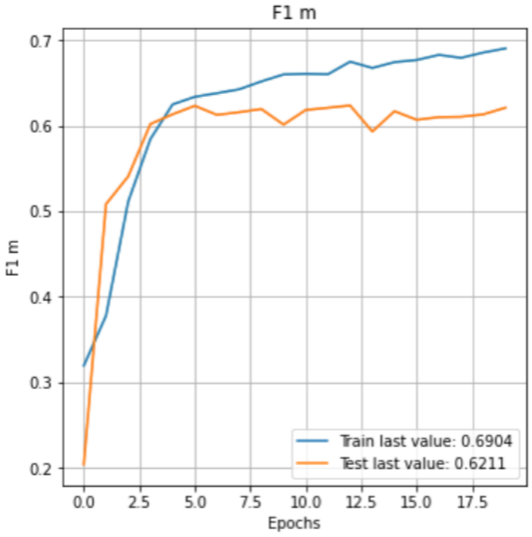

Fig. 13. Training and validation F1 score over epochs :CNN

It is observed that the F1-Score of the prediction of the refactoring type given comments ranges from 61 % to 73%. The Metric we have considered important is the F1 measure, because its the weighted average of the precision and recall, which makes it a reliable metric for an unbalanced data set like ours, unlike accuracy which depends on a balanced data set. Moreover,the result indicates that the Random Forest (RF) and Multi Layer Perceptron (MLP) outperformed all other classifiers in sense of F measure.

RF classifier achieved an F-measure of 73% . MLP classifier achieved an F-measure of 73% as well. These results as well as the performance of each classifier can be seen in Table 4. The class-wise performance of RF can be seen across Figures 14, 15, 16 and 17.

Therefore, to conclude our analysis, the RF and MLP are the best classifiers among the other classifiers in the case of binary classification we are tackling.

We notice that the accuracy are not sufficiently high for most of the models used; this could question the correctness of the refactoring identification. This issue can stem from the fact that the tool utilized to create the data set as it may be unable to recognize all code refactoring labels. Also, by looking at the precision and recall values we notice that the models suggests irrelevant refactoring (false positives) and may miss some true refactoring (false negatives).

But interestingly the deep learning models of LSTM and CNN did rather well, which made us want to investigate into the CNN model further. Figures 11, 12 and 13 showcases the CNN performance epoch-wise in terms of binary classification, loss value during training and F1 measure, respectively. We see a good amount of convergence for the testing and training values meaning that the model is learning well.

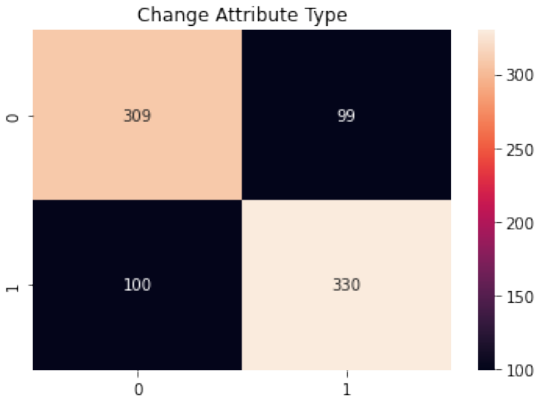

Fig. 14. Change Attribute Type, RF confusion matrix

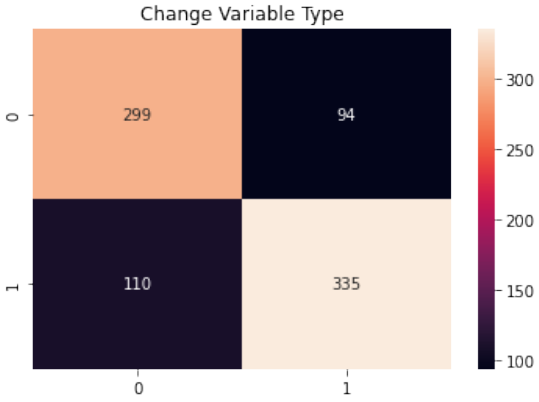

Fig. 15. Change Variable Type, RF confusion matrix

Fig. 16. Rename Class, RF confusion matrix

Model	F1 score	MLSMOTE	Accuracy	MLSMOTE	Time
RF	**0.73**	0.68	0.46	0.36	1.5 min
LR	0.71	0.67	0.40	0.30	9.21 min
SVM	0.66	0.65	0.32	0.33	8.92 min
MNB	0.67	0.66	0.35	0.29	1.2 min
MLP	**0.73**	0.69	0.46	0.34	33 min
CNN	0.62	0.61	**0.73**	0.70	7.33 min
LSTM	0.61	0.60	**0.71**	0.71	125 min

TABLE IV
RESULT OF THE CLASSIFICATION MODELS

Model	Random and Grid Search	F1 score	Custom function	F1 score
RF	n_estimators= 300 min_samples_split=2 min_samples_leaf=1 max_features=2 bootstrap=False	0.73	n_estimators= 1000 min_samples_split=2 min_samples_leaf=1 max_features=auto bootstrap=True	0.74
MLP	hidden_layer_sizes=(100 activation='relu' max_iter=300	0.73	hidden_layer_sizes=(50,50,50) activation='relu' max_iter=400	0.74

TABLE V
THE RESULT AFTER APPLYING HYPER-PARAMETER TUNING

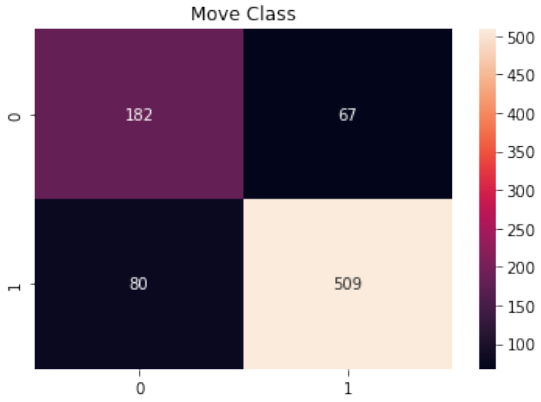

Fig. 17. Move Class, RF confusion matrix

Results of MLSMOTE

From table 5, we witness an unexpected drop in performance, for our models. This can be because, since MLSMOTE is an advanced data oversampling technique in machine learning, and in addition even though our data was imbalanced, but it was probably not to the degree where a method like MLSMOTE was necessarily required. This combination of factors could have potentially led our models to over-fit on the training data and hence, have lower test performances.

A. *Hyper-parameter tuning*

Having seen from the set of machine learning model experiments, we found that the Random forest and Multi-layer Perceptron models had the better performance for our binary classification approach. We felt that this performance can be improved further for this model. Hyper-parameter Tuning is a fundamental and popular approach when it comes to bettering the model itself and its working.We have utilized in our project two of the most common methods are random search and grid search. In addition, we have built a custom function from scratch.

*1) **Random Search and Grid Search**:* Grid search is a great approach for checking every value combination from a predefined list and evaluating the result of each combination. On the other hand ,the Random Search method tries random combinations of a range of values to discover new hyper-parameter combinations to find the best parameter solution. We have combined the two methods together to find the best parameter that will enhance the predictions' results. First, we utilized a random search with a wide range of parameters. Second, the best parameters found by the random search will be used as an input to the Grid search. This approach helps to reduce the execution time of the Grid search.

*2) **Custom function**:* We have tried another solution to find the best parameters for our two models that achieved the best results RF and MLP.We built our custom function which will iterate through predefined parameter values of a specific model and try every combination. The function will evaluate and present the Accuracy, Recall, Precision, and F1 Scores metric for each combination. The custom function is unique because not only have we manually iterated through each parameter of the RF model, but also written a Classifier Chain feature that calculates the varying performance of the model with changing hyper-parameters.

B. *Statistical Significance Test*

Before concluding on our final results, we wanted to conduct a few statistical tests, to analyse the soundness of our results

and its consistency. As well perform those against the other model performances as well. It should also be noted that results have been analysed after having run the models, each on average of 10 runs to make sure that performance remained consistent over time.

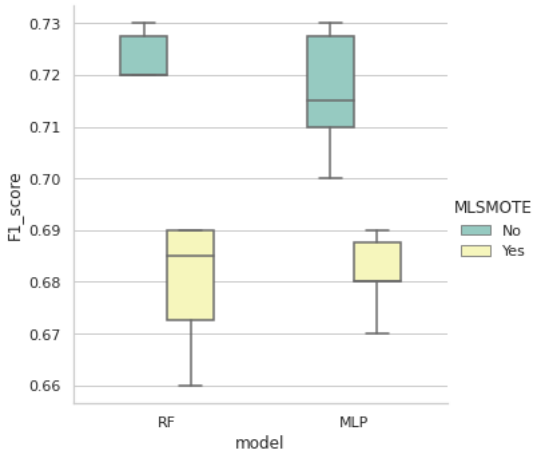

Fig. 18. boxplot diagram for RF (left) and MLP (right)

In Figure 18, we have performed Mann Whitney U test on the two models RF and MLP which are the best models based on their F1 score . The Pvalue is **0.1697** which means the the difference between the randomly selected value of RF and the MLP populations is not big enough to be statistically significant. Next, we performed the same statistical test on the two models RF and MLP after the data is balanced using MLSMOTE method.The Pvalue is **0.9354** which also means the value is not big enough to be statistically significant. Finally, we performed the Mann Whitney U the on the same models before and after we applied the MLSMOTE. We have found the of the Pvalue of RF before and after we applied the MLSMOTE is **0.0001121**, and for MLP is **0.0001333**. These last two values means the difference between the randomly selected values is big enough to be statistically significant.

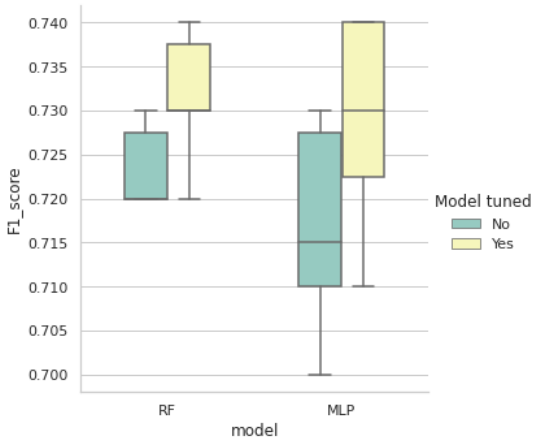

Fig. 19. boxplot diagram for RF and MLP after Hyper-parameter tuning

Here in Figure 19, we have performed Mann Whitney U test on the two models RF and MLP best models based on their F1 score after we applied hyper parameters tuning.The Pvalue is **0.1697** which means the the difference between the randomly selected value of RF and the MLP populations is not big enough to be statistically significant. Next, we have performed the same statistical test on the same models before and after we applied the hyper parameters tuning. We found the Pvalue of RF before and after we applied the hyper parameters tuning is **0.017**, and for MLP is **0.01957**. These values means the difference between the randomly selected values is big enough to be statistically significant.

VII. Future Work

Our future work includes the following:

A. *Potentially better text-processing techniques*

We have applied a number of powerful text-processing methods and that has given a good initial set of results with our models. We feel this can be further made better with using more powerful or more applicable set of NLP-centric pre-processing techniques. Hence, for the future set of experiments with our classification models, we plan to pre-process that data again using Tokenization for better feature extraction and more powerful lemmatization implementation using those tokens and also the Word2Vec architecture for lemmatization.

B. *Research into our deep learning models*

The rather good performance of the CNN model on accuracy and a decent performance on F1 showcases that in-spite of possible wrong or missing labels and even unbalanced data, it is generalising well to the data. This behaviour is rather interesting and we will be investigating further into the parameters that the model is learning along with class-wise weight distribution.

VIII. Conclusion

The main objective of the project was to determine the the type of refactoring that can help solve an SATD comment in need of refactoring, and then automating it. This will make future developments and changes needed to be made, efficient and relevant. This will be particularly helpful in projects and organisations with big developer teams and code-bases that are difficult to maintain or make changes in. The primary goal of this project was to build and leverage the machine learning models and training the models with a valid data set for this purpose. For the immediate experimental studies, we have conducted a series of experimental studies using architectures of Random Forest, Logistic Regression, Multinomial Naive Bayes And Support Vector Machine. As well as deep learning models of Multi Layer Perceptron, Convolutional Neural Network and Long Short Term Memory.

Based on the results of these models, we have seen that MLP and RF performed the best among other classifiers with a 0.73 F1 Score which is a good start, for the initial

study towards predicting refactoring labels. We will look at methods where the current architecture can be further improved towards better performance and accuracy. This form of study will support developers in this role with automatic suggestion tools to identify if the refactoring issue or not, this tool will help to reach the best quality of the software development process.

With our current progress, we have been successfully able to solve a principle problem in tackling SATD. The second and more challenging problem that remains is to be able classify if a given SATD comment requires refactoring or not which has not been delved into before. We will now proceed to continue to polish our current classifier model towards refactoring labels whilst also formulating a statistical architecture towards solving the latter problem.

REFERENCES

[1] N. A. Ernst, S. Bellomo, I. Ozkaya, R. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt,"Proceedings of the 2015 10th Joint Meeting onFoundations of Software Engineering, 2015.

[2] W. Cunningham, "The wycash portfolio management system,"OOPS Messenger, vol. 4, pp.29–30, 1992.

[3] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 91–100.

[4] E. D. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, "An empirical study on the removal of self-admitted technical debt," in2017 IEEE International Conference onSoftware Maintenance and Evolution (ICSME), 2017, pp. 238–248.

[5] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. Newman, A. Ouni, and M. Kessentini,"How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation,"Expert Systems with Applications, vol.167, p. 114176, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095741742030912X

[6] Kaur, Amandeep and Kaur, Manpreet, "Analysis of code refactoring impact on software quality,"MATEC Web of Conferences, vol. 57, p. 02012, 2016. [Online]. Available:https://doi.org/10.1051/matecconf/20165702012.

[7] F. Zampetti, A. Serebrenik, and M. D. Penta, "Automatically learning patterns for self-admitted technical debt removal,"2020 IEEE 27th International Conference on Software Analysis, Evo-lution and Reengineering (SANER), pp. 355–366, 2020

[8] M. Mondal, C. K. Roy, and K. A. Schneider, "A survey on clone refactoring and tracking," Journal of Systems and Software, vol. 159, p. 110429, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219302031

[9] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," Journal of Systems and Software, vol. 107, pp. 1–14, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121215001053

[10] A. S. Nyamawe, H. Liu, N. Niu, Q. Umer, and Z. dong Niu, "Automated recommendation of software refactorings based on feature requests," 2019 IEEE 27th International Requirements Engineering Conference (RE), pp. 187–198, 2019.

[11] A. S. Nyamawe, H. Liu, Z. Niu, W. Wang, and N. Niu, "Recommending refactoring solutions based on traceability and code metrics," IEEE Access, vol. 6, pp. 49 460–49 475, 2018.

[12] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," Journal of Systems and Software, vol. 80, no. 4, pp. 571–583, 2007, software Performance. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412120600197X

[13] R. Krasniqi and J. Cleland-Huang, "Enhancing source code refactoring detection with explanations from commit messages," in 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 512–516.

[14] J. Rennie, R. Rifkin, and A. Memo, "Improving multiclass text classification with the support vector machine," 01 2002.

[15] E. Frank and R. R. Bouckaert, "Naive bayes for text classification with unbalanced classes," in Knowledge Discovery in Databases: PKDD 2006, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 503–510.

[16] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," Journal of Systems and Software, vol. 80, no. 4, pp. 571–583, 2007, software Performance. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412120600197X

[17] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," ArXiv, vol. abs/1807.10059, 2018.

[18] Erin Lim, Nitin Taksande, and Carolyn Seaman. A balancing act: What software practitioners have to say about technical debt. IEEE Softw., 29(6):22–27, November 2012.

[19] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, ser. ICSME 14. USA: IEEE Computer Society, 2014, p. 91100. [Online]. Available: https://doi.org/10.1109/ICSME.2014.31

[20] Francisco Charte and Antonio J. Rivera and María J. del Jesus and Francisco Herrera , "MLSMOTE: Approaching imbalanced multilabel learning through synthetic instance generation," https://doi.org/10.1016/j.knosys.2015.07.019

[21] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. J. Mach. Learn. Res. 12, null (2/1/2011), 2825–2830.

[22] Akhter, M. P., Jiangbin, Z., Naqvi, I. R., Abdelmajeed, M., Mehmood, A., Sadiq, M. T. (2020). Document-level text classification using single-layer multi size filters convolutional neural network. IEEE https://ieeexplore.ieee.org/abstract/document/9016261

[23] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In Proceedings of the 40th International Conference on Software Engineering (ICSE '18). ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/3180155.3180206

[24] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig, "Accurate and Efficient Refactoring Detection in Commit History," 40th International Conference on Software Engineering (ICSE 2018), Gothenburg, Sweden, May 27 - June 3, 2018.

[25] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig, "RefactoringMiner 2.0," IEEE Transactions on Software Engineering, 2020.