



**BIRZEIT UNIVERSITY**

Faculty of Engineering and Technology  
Master of Software Engineering

THESIS

---

**Self-admitted Technical Debt Identification From Source Code Comments  
and Commits Using NLP and Machine Learning.**

---

Author: Ahmed Sabbah

Supervisor: Dr. Abualsoud Hanani

July 28, 2021



**BIRZEIT UNIVERSITY**

**Faculty of Engineering and Technology**  
**Master of Software Engineering**

Self-admitted Technical Debt Identification From Source Code Comments and  
Commits Using NLP and Machine Learning.

تحديد الدين الفني المعترف به ذاتيا من تعليقات الشيفرة المصدرية باستخدام معالجة اللغة  
الطبيعية والتعلم الآلي

**Committee:**

Dr. Abualsoud Hanani , Birzeit University.

Dr. Ahmad Abusnaina , Birzeit University.

Dr. Anas Toma , An-Najah National University.

*A thesis submitted in fulfilment of the requirements  
for the degree of Masters in Software Engineering*

July 28, 2021



---

Self-admitted Technical Debt Identification From Source Code Comments and  
Commits Using NLP and Machine Learning.

---

*Thesis*

Author : Ahmed Sabbah

**Approved by the thesis committee:**

Dr. Abualsoud Hanani : (Chairman of the Committee)

*Abualsoud Hanani*

---

Dr. Ahmed Abusnaina : (Member)

*Ahmad Abusnaina*

---

Dr. Anas Toma : (Member)

*Anas Toma*

---

Date approved:

June 20, 2021

## *Declaration of Authorship*

I, Ahmed Sabbah , declare that this thesis titled, “Self-admitted Technical Debt Identification From Source Code Comments and Commits Using NLP and Machine Learning.” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master degree at Birzeit University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

## *Abstract*

Technical Debt (TD) is a metaphor that describes the immature software that contains the issues are occurred by the developers intentionally. This usually happens when they postponed the optimal code implementation to the future to achieve short-term benefits. TD has been unintentional since the lack of developers' knowledge. Recently, self-admitted technical debt (SATD) term is introduced to express the TD that occurs intentionally. The developers write the comments in the source code file to admit the TD by themselves. Previous studies have shown that the automatic detection of SATD can be done from the source code comments. Most of these studies have focused on the syntactic or pattern analysis of the comments, and little of them considered the sentimental analysis of the comments sentences. This thesis investigates the effectiveness of Natural Language Processing (NLP), machine learning, and deep learning techniques for automatically identifying SADT from source comments and commits. NLP involves pre-processing the source code comments, which are defined as SATD, then extract features from the comments using count-based and word embedding representations. For the count-based approach, TF-IDF method was used. The word embedding for features extracted using pre-trained models was trained on large universal vocabularies such as word2vec, GloVe, BERT, FastText, and specific software engineering models. In the last step, classical machine learning (ML) algorithms will be used, such as Naive Bayes (NB), Random Forest (RF), and Support-Vector Machines (SVM). Additionally, the state-of-the-art neural network techniques that are known as deep learning (DL), such as Convolutional Neural Network (CNN) are used to classify the comments and commits into five classes representing the types of SATD. The types of SATD include requirement debt, design debt, defect debt, test debt, and documentation debt. To achieve that, more than one dataset is used in previous studies was combined.

5082 comments and commits were collected that classified as SATD, but they are not labeled according to the considered five SADT categories, so the manually annotate 1147 comments and 366 commits into one of these five categories. Adataset includes a total of 1513 comments and commits classified to the five categories of SATD . Additionally, another dataset consisting of 4,071 comments were manually labeled and publicly available by the author of [59] (Mdataset). The proposed system can classify the SATD comments and commits into the five categories using classical ML and DL with Adataset, Mdataset, and combined datasets. The RF classifier achieved an accuracy of 0.822 with Adataset, 0.820 with Mdataset, and 0.826 with the combined dataset. With Adataset the CNN achieved an accuracy of 0.838 with BERT. With Mdataset, the CNN achieved an accuracy of 0.809 and 0.812 with BERT and Word2Vec, respectively. Finally, the CNN reached the best accuracy of 0.849 with BERT when using the combined dataset. AS a result, the proposed systems outperform the results of similar studies, which classify the SATD into two types (Requirement and design) by at least 0.11 when using the Mdataset.

## *Acknowledgements*

At the end of the stage, you should close your eyes and take time to remember those wonderful who stood by us to thank them. First, I am thankful and grateful to God for giving me the direction, strength, patience, and opportunity to complete my master's degree. I would like to thank my supervisor, Dr. Abualsoud Hanani for his invaluable advice, continuous support, and patience during my master thesis preparation, and for teaching me how to overcome limitations. I would also like to thank my wife, whom without this would have not been possible. I also appreciate all the support I received from my parents and the rest of my family.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	4
1.2	Problem statement . . . . .	5
1.3	Research contributions . . . . .	5
1.4	Research Questions . . . . .	6
1.5	Structure of thesis . . . . .	7
<b>2</b>	<b>Background</b>	<b>8</b>
2.1	Technical debt . . . . .	8
2.1.1	Self-admitted technical debt . . . . .	9
2.2	Text classification . . . . .	10
2.2.1	Natural Language Processing (NLP) . . . . .	10
2.2.2	Text Pre-processing . . . . .	11
2.2.3	Word representation . . . . .	12
2.2.3.1	Bag Of Word (BoW) . . . . .	13
2.2.3.2	Term Frequency Inverse Document Frequency (TF-IDF) . . . . .	13
2.2.3.3	Word embeddings . . . . .	14
2.2.3.4	Word2Vec . . . . .	15
2.2.3.5	Glove . . . . .	16



2.2.3.6	Fasttext . . . . .	17
2.2.3.7	BERT . . . . .	17
2.3	Machine Learning (ML) . . . . .	19
2.3.1	Statistical classification . . . . .	19
2.3.2	Functional classification. . . . .	20
2.3.3	Neural network classification . . . . .	22
2.3.3.1	Artificial neural network (ANN) . . . . .	22
2.3.3.2	Activation functions . . . . .	24
2.3.3.3	Convolutional Neural Network (CNN) . . . . .	26
<b>3</b>	<b>Literature review</b>	<b>30</b>
3.1	Technical Debt Metaphor: Definition and Expansion . . . . .	30
3.2	Identification of Technical Debt through code-base . . . . .	34
3.3	Identification of Technical Debt through Source Code Comments (Self-Admitted Technical Debt) . . . . .	36
3.3.1	Pattern-based approaches . . . . .	40
3.3.2	Machine-learning approaches . . . . .	42
3.3.3	Deep-learning approaches . . . . .	45
3.4	Identification of Self-Admitted Technical Debt Using Commits Mes- sages . . . . .	47
<b>4</b>	<b>Research Methodology</b>	<b>50</b>
4.1	Dataset description . . . . .	51
4.1.1	Source code comments . . . . .	51
4.1.2	Commits messages . . . . .	52
4.1.3	Manual annotation . . . . .	53
4.1.3.1	Manual annotation result . . . . .	59
4.1.3.2	Kappa Test . . . . .	60

4.1.4	Data exploratory and analysis . . . . .	63
4.1.4.1	Data analysis at text level . . . . .	63
4.1.4.2	Data analysis at features level . . . . .	67
4.2	Research approach . . . . .	70
4.3	System design. . . . .	70
4.3.1	Preprocessing. . . . .	70
4.3.1.1	Tokenization. . . . .	71
4.3.1.2	Text cleaning: . . . . .	72
4.3.1.3	Normalization: . . . . .	72
4.3.2	Features engineering . . . . .	73
4.3.2.1	Syntactic vectorization methods . . . . .	74
4.3.2.2	Word embedding method . . . . .	76
4.3.3	Machine learning Classifiers . . . . .	78
4.3.3.1	Support Vector Machines classifier (SVM) . . . . .	78
4.3.3.2	Naive Bayes classifier (NB) . . . . .	78
4.3.3.3	Random Forest (RF) . . . . .	79
4.3.3.4	Convolution Neural Network (CNN) . . . . .	79
4.4	Evaluation metric: . . . . .	80
<b>5</b>	<b>Experimental setup</b>	<b>83</b>
5.1	Environment setup: . . . . .	85
5.2	Pre-Processing . . . . .	85
5.2.1	Tokenization . . . . .	86
5.2.2	Text cleaning . . . . .	86
5.2.3	Normalization . . . . .	86
5.3	Features engineering . . . . .	86
5.3.0.1	TF-IDF vectorization . . . . .	87

5.3.0.2	Word2Vec vectorization . . . . .	88
5.3.0.3	Universal sentence encoder . . . . .	88
5.3.0.4	GloVe vectorization . . . . .	89
5.3.0.5	Fasttext vectorization . . . . .	89
5.3.0.6	BERT vectorization . . . . .	89
5.3.1	Parameters setting for classifiers . . . . .	89
5.3.1.1	Classic machine learning classifiers . . . . .	89
5.3.1.2	Deep learning classifiers . . . . .	90
<b>6</b>	<b>Experiments and results</b>	<b>91</b>
6.1	Experiments using classical machine learning algorithms . . . . .	91
6.1.1	Experiment set 1: Adataset . . . . .	91
6.1.2	Experiments set 2: Mdataset . . . . .	94
6.1.3	Experiments set 3: The combined dataset . . . . .	96
6.2	Deep learning . . . . .	98
6.2.1	Single-layer CNN . . . . .	99
6.2.1.1	Experiments set 4: Adataset . . . . .	99
6.2.1.2	Experiments set 5: Mdataset . . . . .	101
6.2.1.3	Experiments set 6: Combined dataset . . . . .	103
6.2.2	Multiple-layer CNN (MLCNN) . . . . .	105
6.2.2.1	Experiments set 7: Adataset . . . . .	106
6.2.2.2	Experiments set 8: Mdataset . . . . .	108
6.2.2.3	Experiments set 9: An exploratory experiment for combined-2 dataset . . . . .	111
6.3	Statistical Test . . . . .	116
6.4	Discussion: . . . . .	118

<b>7 Conclusion and future work:</b>	<b>123</b>
7.0.1 Future work . . . . .	125
7.0.2 Threats to validity . . . . .	126
<b>8 Appendix A</b>	<b>136</b>
8.1 Database ER-Diagram . . . . .	136
8.2 Website Pages . . . . .	136

## List of Figures

2.2.1 Word2Vec models architectures [45] . . . . .	16
2.3.1 Support vector machine for linear classification . . . . .	21
2.3.2 ANN with four input neurons and b input called a bias that allows the model to fit the data better,one hidden layer consisting of three neurons, and one target neuron output . . . . .	23
2.3.3 Sigmoid activation function . . . . .	25
2.3.4 Tanh activation function . . . . .	25
2.3.5 ReLU activation function . . . . .	26
2.3.6 Convolutional Neural Networks for Sentence Classification by Yoon Kim. [32] . . . . .	27
2.3.7 Example convolution operation in 2D using a 3x3 filter. The filter move one step for each operation and add the result in the feature map. then, the filter slide to the right and perform the same operation down. . . . .	27
2.3.8 Matrix with stride 2 and 2x2 window . . . . .	28
2.3.9 Max pooling . . . . .	28
2.3.10The averages rounded to the nearest integer. . . . .	29
2.3.11Sum pooling. . . . .	29
3.0.1 literature review workflow . . . . .	31

3.1.1 Technical Debt Quadrant [21] . . . . .	33
4.0.1 Research methodology workflow . . . . .	50
4.1.1 Sample of comments dataset [59] . . . . .	51
4.1.2 Sample of commits dataset; 1 classified as SATD, 0 unclassified [54] . . . . .	53
4.1.3 Comments classification . . . . .	59
4.1.4 Commits classification . . . . .	59
4.1.5 Experts versus author for requirements classification . . . . .	60
4.1.6 Experts versus author for Design classification . . . . .	61
4.1.7 Experts versus author for Defect classification . . . . .	61
4.1.8 Experts versus author for Test classification . . . . .	62
4.1.9 Experts versus author for Documentation classification . . . . .	62
4.1.10 Input data for Kappa test . . . . .	62
4.1.11 Number of characters appearing in each comment without pre- processing . . . . .	64
4.1.12 Number of words appearing in each comment without preprocess- ing . . . . .	64
4.1.13 Number of words appearing in each comment after preprocessing	65
4.1.14 Average word length appearing in each comment . . . . .	65
4.1.15 Frequency of stop words . . . . .	66
4.1.16 Average word length appearing in each comment after stop words removed . . . . .	66
4.1.17 Frequency of words . . . . .	67
4.1.18 Frequency of words after remove punctuation marks . . . . .	67
4.1.19 Word cloud data representation . . . . .	68
4.1.20 Co-occurrence network of words . . . . .	69

4.3.1 System Design . . . . .	71
4.3.2 BOW vectors for tow comments . . . . .	74
4.3.3 BOW after applying N-gram range between 1 and 3 . . . . .	75
4.3.4 TF-IDF representation for two comments . . . . .	76
4.3.5 Word2vec models for different domain . . . . .	77
4.3.6 The three main processes in system design . . . . .	80
4.4.1 confusion matrix for the five SATD classes . . . . .	81
6.1.1 RF and TF-IDF performance metrics for each type of SATD in Adataset . . . . .	92
6.1.2 The accuracy of classic machine learning with TF-IDF and USE for Adataset . . . . .	93
6.1.3 RF and TF-IDF performance metrics for each type of SATD in Mdataset . . . . .	94
6.1.4 The accuracy of classic machine learning with TF-IDF and USE for Mdataset . . . . .	96
6.1.5 RF and TF-IDF performance metrics for each type of SATD in combined dataset . . . . .	97
6.1.6 The accuracy of classic machine learning with TF-IDF and USE for combined dataset . . . . .	98
6.2.1 SLCNN and BERT performance metrics for each type of SATD in Adataset . . . . .	99
6.2.2 The accuracy of single-layer CNN with TF-IDF and pre-trained models for Adataset . . . . .	101
6.2.3 SLCNN and Word2Vec performance metrics for each type of SATD in Mdataset . . . . .	102

6.2.4 The accuracy of single-layer CNN with TF-IDF and pre-trained models for Mdataset . . . . .	103
6.2.5 SLCNN and Word2Vec performance metrics for each type of SATD in combined dataset . . . . .	104
6.2.6 The accuracy of single-layer CNN with TF-IDF and pre-trained models for combined dataset. . . . .	105
6.2.7 MLCNN and BERT performance metrics for each type of SATD in Adataset . . . . .	106
6.2.8 The accuracy of multiple-layer CNN with TF-IDF and pre-trained models for Adataset. . . . .	107
6.2.9 MLCNN and SLCNN results for Adataset . . . . .	107
6.2.10Chart MLCNN and SLCNN results for Adataset . . . . .	108
6.2.11MLCNN and BERT performance metrics for each type of SATD in Mdataset . . . . .	108
6.2.12The accuracy of multiple-layer CNN with TF-IDF and pre-trained models for Mdataset. . . . .	110
6.2.13MLCNN and SLCNN results for Mdataset . . . . .	110
6.2.14Chart MLCNN and SLCNN results for Mdataset . . . . .	111
6.2.15Exclude on merge four strategy . . . . .	114
6.2.16The accuracy of all classifiers with TF-IDF,USE and pre-trained models for combined-2 dataset. . . . .	116
6.3.1 Nemenyi test Shows the p values for each pair . . . . .	118
6.4.1 . . . . .	121
6.4.2 Confusion matrix for MLCNN with BERT model for Adataset . .	121
6.4.3 Confusion matrix for RF with TF-IDF for Mdataset . . . . .	122
8.1.1 Database ER-Diagram . . . . .	136



8.2.1 Home Page part 1 . . . . .	137
8.2.2 Home Page part 2 . . . . .	137
8.2.3 Home Page part 3 . . . . .	138
8.2.4 Information of participant . . . . .	138
8.2.5 Classification page . . . . .	139

## List of Tables

3.1	Summary papers that related to this thesis . . . . .	49
4.1	Number of new collected comments . . . . .	52
5.1	Environment setup . . . . .	85
5.2	Unique words with special terms . . . . .	87
5.3	Unique words without special terms . . . . .	87
6.1	ML performance metrics with all classifiers for Adataset . . . . .	92
6.2	ML performance metrics with all classifiers for Mdataset . . . . .	95
6.3	ML performance metrics with all classifiers for combined dataset . . . . .	97
6.4	SLCNN performance metrics with all pre-trained models and TF-IDF for Adataset . . . . .	100
6.5	SLCNN performance metrics with all pre-trained models and TF-IDF for Mdataset . . . . .	102
6.6	SLCNN performance metrics with all pre-trained models and TF-IDF for combined dataset . . . . .	104
6.7	MLCNN performance metrics with all pre-trained models and TF-IDF for Adataset . . . . .	106
6.8	MLCNN performance metrics with all pre-trained models and TF-IDF for Mdataset . . . . .	109

6.9	ML classifier for combined-2 dataset after exclude requirements	
	TD from Mdataset . . . . .	114
6.10	SLCNN for combined-2 dataset after exclude requirements TD	
	from Mdataset . . . . .	115
6.11	MLCNN for combined-2 dataset after exclude requirements TD	
	from Mdataset . . . . .	115
6.12	Average ranking sorted descending for all classifiers . . . . .	117

# Chapter 1

## Introduction

The software industry faces many conflicting goals that must be dealt with, such as delivering in a short period, software maintenance, high-quality software, and on budget. These goals lead the developers to workarounds, make wrong or unhelpful technical decisions to submit their work. The short-term plans are usually satisfied, but the long-term are impacted negatively with an increased cost when be paid off [23]. Thus, in practice, the developer starts the technical debt life.

Technical Debt (TD) is a metaphor coined by Ward Cunningham [12]. It reflects the additional cost that implies rework caused by a sub-optimal solution instead of using the better software development life cycle approach. The concept of TD is derived from financial debt, as the interest resulting from the late payment. Like the financial debt, TD has an interest, and the cost increases if not pay the debt early by refactoring the code at the appropriate time to avoid interest in the future.

Technical debt is extremely correlated with immature software and issues in software development, such as requirement debt that measures the difference between the requirement specification and the actual software implementation.

Design debt refers to the violation of the good design principle, where code debt includes poor maintenance and readability that needs to be refactored. Documentation debt which expresses on the lack of information that describes the code, and finally test debt that describe the shortage of accepted testing [12]. Some previous studies have presented that technical debt is spreading widely in software, which is inevitable and may have an effect on the software quality[36].

The developers' accumulation of technical debt may be deliberately or inadvertently. Often it was inadvertently [47]. Inadvertently TD occurs when the developers afford the debt without intentional, for example, when the developer writes a code with low quality because of insufficiency of experience. Furthermore, deliberately TD occurs with the intention of developers, in a particular situation when the project manager decides to release the software early. When the developers admit these issues and are documented often by comments in the source code files, technical debt takes the name 'Self-Admitted Technical Debt' (SATD). This term was first coined by Potdar, and Shihab [53]. It is a technical debt that is written by the developers deliberately, through comments or commits messages, with the knowledge that the implementation is not an optimal solution for the software.[28]

Recently, and after the self-admitted technical debt term introduced [28], researchers focused on SATD in more than one direction. Most of the directions can be reduced to three categories, as have introduced in [58]. The first direction is detection: which focuses on identifying or detecting the SATD in source code comments. The second is comprehension: those studies focus on the relation of SATD with different aspects of the software process. The third is repayment, which includes the studies with the aim to investigate tools and techniques to remove 'fully repay' or mitigate 'partially repay.' Wehaibi et al. [63] assert that the

proportion of SATD in the project may have a negative effect on the complexity of software. In addition, they discovered that files of the source code that included self-admitted technical debt have more bug fixing changes, whereas files that not contains SATD have more defects.

Some other studies take into consideration the commit messages to investigate the effect and relation with SATD. Yan et al. [68] introduced the level of change in self-admitted technical debt determination. This model determines whether the change introduces SATD by using the versions of the code comments, identifying the SATD at file level for each version, and analyzing and extracting information from a message in commits written by the developers to predict if the commit is related to SATD.

This thesis aims to develop an automatic system that identifies the SATD comments and commits that are written by the developer. The system classifies the SATD comments into five categories: defect, design, documentation, requirement, and test. This study investigates the effectiveness of Natural Language Processing (NLP), machine learning, and deep learning techniques for automatically identify SATD from source comments and commits. The NLP involves pre-processing the source code comments defined as SATD, then extracting the features from the comments using two approaches; count-based and semantic representative. For the count-based approach, the TF-IDF method was used. The semantic-based features extraction uses a word embedding approach and pre-trained models trained in large universal vocabularies such as word2vec, GloVe, BERT, Fasttext, and specific software engineering models. In the last step, the classic machine learning (ML) algorithms were used, which include: Naive Bayes (NB), Random Forest(RF), and Support-Vector Machines (SVM). Additionally, state-of-the-art neural network techniques known as deep learning (DL) will be used that include Convolutional Neural Network (CNN),

aiming to classify the comments into five classes indicating the type of SATD. The types of SATD include requirement debt, design debt, defect debt, test debt, and documentation debt.

## 1.1 Motivation

Software quality is one of the most critical aspects for the software company and customers. The development team has various ways to ensure the quality of software for different purposes. One of these purposes is to measure how the software compatible with the software engineering principle at the code-base level by using static analysis of source code. Code smell, bad code, dirty code, technical debt, and other terms appear in software development fields, indicate the immature software and need more effort and work to increase the quality of the software. For example, when the developer writes a bad code or violates the design principle, it is better to document the violation by self-admitted technical debt forward to refactor the source code in the future. There are some tools that are used to analyze the source code to find the violation and code smell. These tools depend on the predefined rules and structure without investigating the logic of the code as humans do. This motivates us to develop an automatic system for identifying the technical debt that the developer admitted by him/herself. Usually, the developer writes comments in natural language. The proposed system uses the Natural Language Processing (NLP) techniques to analyze and understand the developer's intents. Then a machine learning classifier identifies the comments to the five most important categories of self-admitted technical debt. With the knowledge gained from the literature review chapter, the motivation of this study is to fill the gap in SATD identification by investigating more effort in the dataset used, NLP techniques, and machine learning approach for

classification SATD. Most studies used the same dataset that was introduced by Maldonado et al. [59] to identify the comments to SATD or not. These studies did not consider the types of SATD. The pre-trained models were used in this study; they trained on universal context and specific software engineering domains and used state-of-the-art deep learning techniques such as Convolutional Neural Network (CNN) with different feature engineering models.

## **1.2 Problem statement**

Self-admitted technical debt is an express written by the developer in the file of source code. It describes the code or part of code written by the developer without considering the optimal way. These technical debts need to manage and remove as possible. The aim of managing the technical debt is to pay the interest on time to avoid more accumulated interest in the future. The first step is to identify the technical debt types to know how to treat them in specific filed that appear, including (Requirement, design, test, defect, and documentation), since the comments that indicate the technical debt are written by using natural language. This leads us to process these comments by using NLP techniques to represent the meaning of the comments sentences. Additionally, machine learning and modern deep learning techniques are adopted to classify the types of SATD.

## **1.3 Research contributions**

This thesis achieved the following contributions.



1. Collected a sufficient dataset with 1758 commits and 3102 comments used in previous studies and extracted 222 comments from two open-source android applications. All the comments and commits are labeled as SATD.
2. Classified the SATD into five types that include (requirement, design, test, defect, and documentation) using a multi-classification model for cross-project. This approach is different from previous studies that used binary classification and classified SATD into only two types(Requirement and design).
3. Manually annotated part of the collected comments with a help of software engineering experts. The comments that belong to the considered five SATD types are used in this study.
4. The NLP word embedding representation techniques were used with pre-trained models such as Word2vec, BERT, Glove, Fasttext, and pre-trained specific model in software engineering context as described in [16].
5. Deep learning technology was used, such as Convolutional Neural Network (CNN) to classify the input comments into the SATD five categories.
6. The proposed system was evaluated by using two datasets, and compared the results with the results published in the previous similar studies.

## 1.4 Research Questions

1. How well the NLP pre-trained models can improve the identification of self-admitted technical debt from source code comments and commits effectively?

2. How well machine learning algorithms that include (SVM, NB, RF, and CNN) can automatically classify the five SATD types efficiently?
3. How well combined the two datasets improve the classification accuracy?
4. Does increasing the numbers of layers in CNN model improve the accuracy of the study approach?

## 1.5 Structure of thesis

The rest of thesis is structured as follows:

**Chapter 2 :** presents the background about main content of study, that include self-admitted technical debt, NLP, deep learning, machine learning, pre-trained models.

**Chapter 3 :** discusses literature review related to technical debt, SATD identification and classification.

**Chapter 4:** provides complete details about the research methodology.

**Chapter 5:** presents the experiment setup, the tools used in experiments, the parameters of classifiers, pre-trained models specification.

**Chapter 6:** presents the experiments results, discussion of the results, and statistical test.

**Chapter 7:** provides conclusion, future works and threats to validity

## Chapter 2

### Background

The Background chapter discusses the definition of technical debt, self-admitted technical debt, and reviews the details of the automatic text classification, including natural language processing and machine learning techniques. Natural language processing has text pre-processing to clean the text and feature extraction to convert the words into vectors, such as bags of word(BoW), TF-IDF, and word embedding. In addition, the word embedding models pre-trained on the billions of words such as Word2vec, Glove, BERT, and Fasttext. Moreover, three types of machine learning are discussed; **statistical classification** such as Naive Bayes classifier, **functional classification** and **neural network classification** that include modern approaches in deep learning classification depending on the neural network such as Convolution Neural Network (CNN).

#### 2.1 Technical debt

Technical Debt (or TD) is a description of immature software because of its containment of the issues that occurred by a developer intentionally, when they postponed fixing the issue to the future, or unintentionally since lack of knowledge. Technical Debt is a metaphor coined by Ward Cunningham [12], and he

considers “not quite right code” is technical debt. The concept of TD is derived from financial debt, as the interest result from the late payment. Like the financial dept, TD has an interest, and the cost increases if not paying the debt early by refactoring the code at the appropriate time to avoid interest in the future.[12]

### 2.1.1 Self-admitted technical debt

Source code comments are explanation or annotation that written by the developers. Comments allow developers to clarify, document and express concerns about implementing an informal method that does not affect the program’s functionality. These comments are generally ignored by compilers and interpreters.[37]

Self-Admitted Technical Debt (SATD) is the source code comments written by a developer in the source code file and indicates an issue in the code. The term self-admitted technical debt is introduced the first time by Potdar and Shihab[53]. There are various types of TD described by the comments (SATD). For example, **design debt** is the comment that indicates long methods, lack of implementation, poor abstraction, misplaced code, and workaround or temporary.

*“TODO: - This method is too complex, lets break it up” - [from ArgoUml] “/\*  
TODO: really should be a separate class \*/” - [from ArgoUml]*

Technical debt, SATD and their types are described in details in chapter 3 ‘Literature review’.

## 2.2 Text classification

Text can be classified in two different ways: manual or automatic classification. Previously, a human read the text, interprets the content, and categorizes it accordingly. This method can usually provide good results, but it is expensive and time-consuming. While, the automatic classification that uses natural language processing and machine learning techniques can be transferring the text classification into new areas fastly, more cost-effective, and more accurate way[13].

The automatic text classification passes in two phases; NLP and machine learning. In the first phase, the text is cleaned and normalized. Then representative feature vectors are extracted from the words of sentences that indicate the relationship between the word meaning and context. The resulting vectors are understandable by the machine. In the machine learning phase, the machine learns how to make the classifications based on past observations. Machine learning algorithms use the pre-labeled data for learning, and they can conclude different associations between pieces of text[37].

### 2.2.1 Natural Language Processing (NLP)

NLP is the artificial intelligence field that makes the computer understand and process the human language by converting the text into a mathematical form that can be treated by the machine [46]. NLP is used in different domains of the industry. This thesis will be focused on the software engineering domain. In any software project, many documents are produced within the project itself, whether before constructing the project, during development or after the

deployment. These documents are written in the human language, such as requirements, design documentation, use-case scenarios, bug reports, user manuals, commit messages, etc. Along with the code comments, the developers depend on all of these text data to build the software from the design to the testing. Lastly, many resources enter the software engineering domain research by analyzing textual information such as stack overflow, app market, user review, etc., to support the development activities. In other words, gain new insights and extract the knowledge[5]. Natural language processing tasks involve syntactic and semantic analysis that is used to make a machine understand the text[46].

**Syntactic analysis** : Also known as syntax or parsing. It analyzes the text with the rules of a formal grammar, and identifies the relations between the words, and represents the text on a diagram such as a parse tree[46].

**Semantic analysis** : Is the process of extracting the meaning from the text sentences , and analyzing the relationship between each word and related context , in attempting to understand and discover the meaning of the text.

### 2.2.2 Text Pre-processing

The pre-processing task is the phase of preparing and cleaning the text data by remove the words and accessories, that do not add any meaning to the text such as: remove tags, numbers, punctuation and stop-words. Additionally, the preparing data include[49].

- **Case folding** : Convert the characters of words to small letters.
- **Tokenization** : Is a way to split the text into smaller units called tokens, it can be either statements, words, sub-words, and characters.
- **Spell Check** : is a checking the style and grammar correctness of a text.[13]

- **Stemming** : Is a method for removing common inflectional endings from words in the English language. Porter stemming algorithm is one of the common algorithms used [52].
- **Lemmatization** : This process is the same stemming, but instead of removing inflectional ending of the words, it uses lexical knowledge bases to get the meaningful base forms of words. The lemmatization gave the best result for information retrieval in comparison study with stemming[6].
- **Contraction** : The way that convert the contraction words to expansion form (i.e. aren't become are not)
- **Parts Of Speech (POS) tagging** : Is the process to identify each word in the sentences to know how a word is used in every sentence (noun, verb, adjective, adverb.. etc).
- **N-grams** : Look the same way to tokenize, but in this way the split of text is more than one size for each token, some words used together to give specific meaning (e.g. calling method). When using these words individually, they carry a completely different meaning. Three common types of N-grams used, unigrams when N equals to one, Bigrams (2-grams), and trigrams (3-grams) [31].

### 2.2.3 Word representation

Word representation is aiming to represent a word in mathematical form, to be understandable by the machine. This process is also known as a feature extraction or vectorization. The followings are the popular methods used for converting the words to numerical vectors:

### 2.2.3.1 Bag Of Word (BoW)

Bag-of-word is a representation model that counts the unique word occurrence frequency in the text document. By this, each sentence can be represented as a vector of numbers of the same length. The length of the vector is the same size of the unique words in the whole dataset. If the word exists in the sentence, the value of the vector is 1, if it duplicated in the same sentence the value is 2, and 0 when the word does not exist [29]. It is possible to apply the N-grams tokenization during building the BoW, and the representation of words could be one word or more. The BoW model is a way to represent the text to numbers. However, this model still has limitations. It can get extremely huge in the vocabulary for the large text. This can lead to a huge size of vectors that represents each sentence, which can result in poor performance. This model does not consider the semantics or meanings associated with a token or phrases in the document. It completely ignores the context in which a word or a phrase is being used, that may capture the meaning from the neighborhood of words.

### 2.2.3.2 Term Frequency Inverse Document Frequency (TF-IDF)

TF-IDF is a common feature extraction technique to represent the text document into a meaningful pattern by transforming the document into numbers which are used to fit the machine learning algorithms. TF is statistical measure to calculate the frequency of terms across the document. It appears like BoW approach with different in accounting method the number of terms frequently in the document. There are several ways for normalizing the frequency. The most common way is dividing number of word occurs in a document by the total number of words in all documents [29]. IDF measures the frequency of words across a set of documents. It is used to calculate the importance of a term



and avoiding the terms that occur frequently without adding importance for the meaning. IDF is calculated as  $IDF(w) = \log(N/DF)$ , where  $N$  is the number of documents and  $DF$  is the number of documents containing word  $w$  [1]. In the IDF just the opposite of TF is done. The weights of the important terms, which have more meaningful but less frequency, are scaled up. On the other hand, the less important terms with more frequency are scaled-down. But, that did not happen in TF where it focused only on the number of times the terms occur frequently in documents without considering the importance of terms. As a result, the weight of word  $W$  in document  $D$  across the text corpus  $T$  is given by the following TF-IDF equations[31]

$$TF_{(W)} = \frac{\text{Number of times the word } W \text{ occurs in a document}}{\text{numbers of words in the document}} \quad (2.1)$$

$$IDF_W = \log \frac{\text{Total numbers of documents}}{\text{Number of documents containing word } W} \quad (2.2)$$

$$weight_{(W,D)} = TF_{(W,D)} \times IDF(W) \quad (2.3)$$

TF-IDF provides an improvement over BoW, by scaling up the weights of the less frequently words but they important. However, it still depends on syntactic analysis without taking into consideration semantics, context associated with words, co-occurrence of terms, and the representation vector is still large for the huge vocabulary of the large text.

### 2.2.3.3 Word embeddings

Word embedding is different from the two discussed vectorization approach, BoW and TF-IDF, word embedding is a language representation models. They

considered the neighborhood of the word, in terms of what words come after or before the specific word. By considering the neighborhood of a word, it captures the important information in terms of what context the word appears in a sentence. This model supports the semantics and syntactic methods, by defining the relationship between the word and neighborhood to discover the context of word in the sentence. This model represents each word in sentences by a vector with N-dimensional space. All the words that have the same meaning should have similar representations. The focus will be on the most three popular techniques of word embedding that are used in the set of experiments; Word2vec, Glove, Fasttext, BERT, and software engineering Word2Vec pre-trained model.

#### 2.2.3.4 Word2Vec

Word2vec was developed by Thomas Mikolov et al. at Google [44]. It is a model that uses contextual information with neighborhood consideration of words as input to produce the word vectors as output. Word2vec is an unsupervised machine learning algorithm used for building pre-trained word embedding models by clustering these word into vectors. It is not only one algorithm, but it includes two learning models, Continuous Bag of Words (CBOW) that predict the target word based on the context word, and Skip-gram predict the context word based on the target word [38]. Figure 3.0.1 show tow models architectures.

The output of the Word2vec algorithm is a  $|V| * D$  matrix, where  $|V|$  is the size of the vocabulary that generated from all words, and  $D$  is the number of dimensions that represent each vector of the word. Each row in the embedding matrix includes an individual word in the vocabulary, the number of dimensions ( $D$ ) can be changed and it relies on several factors. In real-world use cases, the  $D$  takes value between 50 and 300[31].

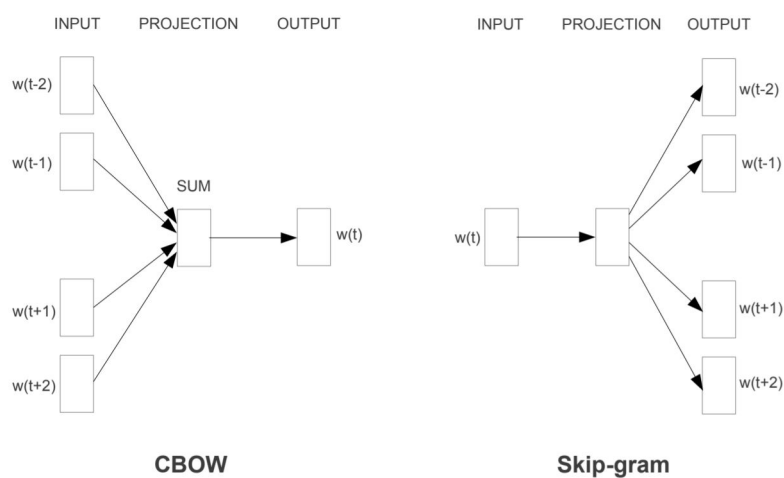


FIGURE 2.2.1: Word2Vec models architectures [45]

There are pre-trained models that use the Word2vec algorithm and provided by researchers in different domains. Google provided a Word2vec model that is trained on a huge Google news dataset. The vocabulary size contains 3 million words and phrases, and each vector has 300 dimensions<sup>1</sup>.

### 2.2.3.5 Glove

GloVe is a word vector technique that stands for Global Vectors introduced by Pennington et. al. [51] It combines two models feature, that involve global matrix factorization and local context window methods such as skip-gram model. The GloVe model uses a co-occurrence matrix to make word embedding. Each word represented in the row of matrix, and the context that the word appears in represented in the column of the matrix (i.e. co-occurrence matrix: words x context). The value of co-occurrence matrix is the count of a word that appears in a given context. Then, the co-occurrence matrix is factorized to reduce the dimension and generate the lower-dimensional embedding matrix, each row is a vector representation for each word. Pre-trained models for Glove available on

<sup>1</sup><https://code.google.com/archive/p/word2vec>

<sup>2</sup>The public domain model “glove.840B.300d” will be used, that includes 840B tokens, 2.2M vocabulary, 300 dimension vectors, and the model size is 2.03 GB.

### 2.2.3.6 Fasttext

Fasttext is the state-of-the-art word embedding approach that working at character-level. Fasttext introduced based on two studies [30], [9]. Which is essentially an extension of word2vec model, but each word is broken down into character n-grams. As a result, a word’s vector is made up of the number of this character’s n-grams. For example, the vector of word “method” is a sum of the vectors of the n-grams characters: “<me”, “met”, “meth”, “metho”, “method>”, “eth”, “etho”, “etho”, “hod”, “hod>”, “od>”. The “crawl-300d-2M.vec” will be used: 2 million word vectors trained on Common Crawl (600B tokens) with 300 dimensional vector <sup>3</sup>

### 2.2.3.7 BERT

BERT is a language representation model, which stands for (Bidirectional Encoder Representations from Transformers) that was introduced by Google [51]. To understand the BERT model, Let’s consider an example.

Survey **Monkey**, is an online survey system.

A **Monkey** is an animal.

**Monkey** testing is a testing methodology, where the input is random values.

---

<sup>2</sup><https://nlp.stanford.edu/projects/glove/>

<sup>3</sup><https://fasttext.cc/docs/en/english-vectors.html>

In all of these sentences, no matter what context Monkey is being used in, and the Monkey will be embedding the same. BERT solves this problem by representing the word based on the current context it is being used in. Additionally, BERT model, built using semi-supervised models to support reuse by fin-tuned for specific task. In other words, a model developed on a broader task and it can be reused as the starting point for a model on a second task. This method is called transfer learning.

BERT was built using Transformer architecture that includes two mechanisms, namely an encoder component to read the text input ,and a decoder that outputs a prediction depending on the type of task such as classifications. BERT uses two unsupervised tasks to build pre-trained models, these strategies are masked language model (MLM) and next-sentence prediction(NSP). The MLM BERT model selects 0.15 of the tokens randomly and masks them. Next, it tries to predict the original value of the masked tokens, based on other context provided. This method is used in the classification task, because it deals with one sentence. However, when tasks involve a pair of sentences, and the needing to relationship between multiple sentences captured, as often used in question and answer tasks. To do this task , BERT applies next-sentence prediction strategies. A pair of sentences, A and B, are input to the BERT model, 0.50 of selected sentences are actually the next sentence for each sentence. B would be actually the next sentence in the original text ,and it comes after A, while in the other 0.50 of sentences selected randomly, and it is not the next sentence after A, and B would be not actually the next sentence in the original text. The NSP model will predict whether sentence B is actually the next sentence following sentence A or not [31].

## 2.3 Machine Learning (ML)

Some of the most popular machine learning algorithms that used in text classification, fall into one of methods that include statistical classification, functional classification and neural classification[39].

### 2.3.1 Statistical classification

The most used type of classification algorithm is the Naive Bayes classifier (NB). Naive Bayes is not only one algorithm but a collection of supervised learning algorithms based on applying Bayes' theorem. These families of algorithms apply a common principle of simple probabilistic classifiers. NB classifier assumes ("Naively") that each feature is being classified as independent on other features<sup>4</sup>. While, "Bayes" theorem finds the probability of the feature, based on previous knowledge of situations that may be correlated with the feature<sup>5</sup>. The Bayes' theorem can be represented as following equation[31] .

$$P(A|B) = \frac{P(B|A).P(A)}{P(B)} \quad (2.4)$$

Where A and B are events:

- $P(A|B)$  is the probability of A given B, while  $P(B|A)$  is the probability of B given A.
- $P(A)$  is the independent probability of A, while  $P(B)$  is the independent probability of B.

---

<sup>4</sup>[https://scikit-learn.org/stable/modules/naive\\_bayes.htm](https://scikit-learn.org/stable/modules/naive_bayes.htm)

<sup>5</sup><https://monkeylearn.com/blog/practical-explanation-naive-bayes-classifier>

Random forest is a supervised machine learning algorithm. It creates a "forest" out of an ensemble of decision trees, which are normally trained using the "bagging" technique. The idea of the bagging method is that a combination of learning models increases the overall result. Random forest has emerged as a promising text categorization tool. Random forest is a common classification method that combines a number of classification trees into a single ensemble. Breiman suggested one of the most common forest construction procedures: randomly selecting a subspace of features at each node to grow decision tree branches, then using the bagging approach to produce training data. Then, using the bagging technique, create training data subsets for individual trees construction, and eventually, combine all individual trees to form a random forest model.[66]

### 2.3.2 Functional classification.

functional or (geometrical) classification, is representation of each document or word as a dot in a multidimensional space where the number of dimensions is equal to the number of features in the feature vector[39].

One of the simplest algorithm k Nearest Neighbours (kNN), it look around the dot that represents the document being classified in the multidimensional space, to find the k number of nearest neighbours to that dot. Then the kNN classifies the new document to the category that most of nearest neighbours belong to.

Support Vector Machines SVM is other text classification algorithm that outperform , it is a supervised machine learning algorithm that try to classify data by finding the optimal hyper plane that segregates between the classes of classification.The SVM is different from other algorithms by the method of finding the decision boundary between the class. The best judgment boundary that

maximizes the distance to the both classes' neighboring data points. Decision boundary also called margin classifier or margin hyper plane. To understand the SVM algorithm let's take the following example.

Suppose have a balls basket with two types of balls in it with two features about the balls ; weight and radius. SVM algorithm will be created to separate between the kinds of balls. Therefore, this problem can be solved by represent each ball as a vector in two dimensional space, and find the best line that segregate the two types of balls as shown in the following diagram 2.3.1. In order to identify a hyper plane , the equation can be as follows[31]:

$$W_1 * radius + W_2 * weight - C = 0 \quad (2.5)$$

Where, W1 and W2 are coefficients and C is a constant.

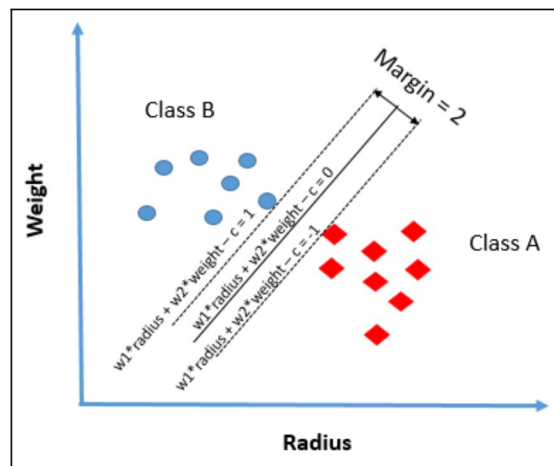


FIGURE 2.3.1: Support vector machine for linear classification

This example shows the simple SVM used for linear regression and classification problems in two dimensional space. Kernel SVM is algorithm that used for linear and non-linear data, when there are more than two features to fit a hyperplane in  $n$  dimensional space. This type commonly recommended for text



classification, and the equation of the hyperplane in the  $n$  dimension with linear data can be generalized as follows:

$$f(X) = W^t * X + c \quad (2.6)$$

Here,  $W$  is a weight vector of coefficients and  $X$ (feature) represents each dimension of the space.

### 2.3.3 Neural network classification

The term of neural network mostly used with deep learning. In fact the deep learning is a evolution of a machine learning, it is a subset of a machine learning. both of them laying under the umbrella of Artificial Intelligence (AI) .The deep learning is designed with aim to analyze data continuously, and simulate the logical structure of human to how draws conclusions [32]. To achieve this objective , deep learning algorithms use a layered design called an artificial neural network that derived from the biological neural network of the human brain.Two types of neural network discuss in following

#### 2.3.3.1 Artificial neural network (ANN)

Artificial Neural Networks (ANN) are a set of Neurons organized in multi layers and fully connected with each others. The synapses that connect the neurons usually have weights that adjust during learning proceeds. ANN also known as a Feed-Forward Neural network, because the data are input into ANN layers are processed only in the forward direction[26].

To understand the ANN let's consider a set of features represented by  $X$  equation 2.7. A set of weights,  $W$  as shown in equation 2.8:

$$X = x_1, x_2, x_3, \dots, x_n \quad (2.7)$$

$$W = w_1, w_2, w_3, \dots, w_n \quad (2.8)$$

Then for each neurons in the hidden layer will apply the following equation:

$$z = x_1.w_1 + x_2.w_2 + x_3.w_3 \dots + x_n.w_n + b \quad (2.9)$$

The ANN boils down in figure 2.3.2.

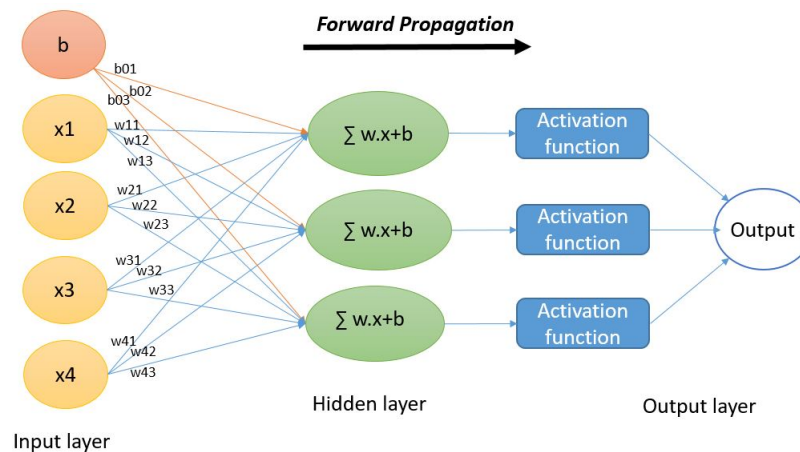


FIGURE 2.3.2: ANN with four input neurons and  $b$  input called a bias that allows the model to fit the data better, one hidden layer consisting of three neurons, and one target neuron output

As shown in figure 2.3.2, three layers used in neural network :

- **Input layer** : In this layer the input is features, and the number of features is equal the number of nodes or neurons that will fed to the network.
- **Hidden layer** This layer is located between input and output layers, and it

can be one or more layers, each layer have number of nodes, the computation done in this layer to extract the pattern of data. Number of layers and nodes considering hyperparameter that need to be tuned, and depending on the complexity of the shape that need to find the decision boundary.

- **Output layer** The output layer provide a result for a particular input. The number of node in this layer depends on the type of classification problem that being solved, in the binary classification only one node needing 0 or 1 to determine to which category the input belong. However, for multi classification the number of nodes must equal to the number of classes.

Before the output layer in neural network model as a final step, the activation function is being calculated to offer a powerful way to find the decision boundary especially for non-linearity equation in any dimensional space. There are various types of activation functions, the most commonly three types described in the following.

### 2.3.3.2 Activation functions

**Sigmoid:** This activation function transforming the output into a range between 0 and 1.[31] It define as following equation 2.10, and the curve shown in figure.2.3.3

$$S(x) = \frac{1}{1 + e^{-x}} \quad (2.10)$$

**Hyperbolic tangent (tanh):** Tahn converts the input values within the range of -1 to 1 [31]. Than equation 2.11, and curve in figure 2.3.4

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (2.11)$$

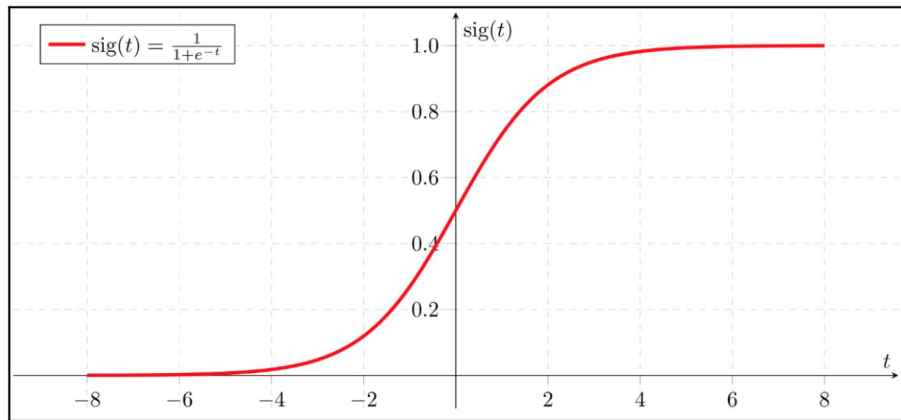


FIGURE 2.3.3: Sigmoid activation function

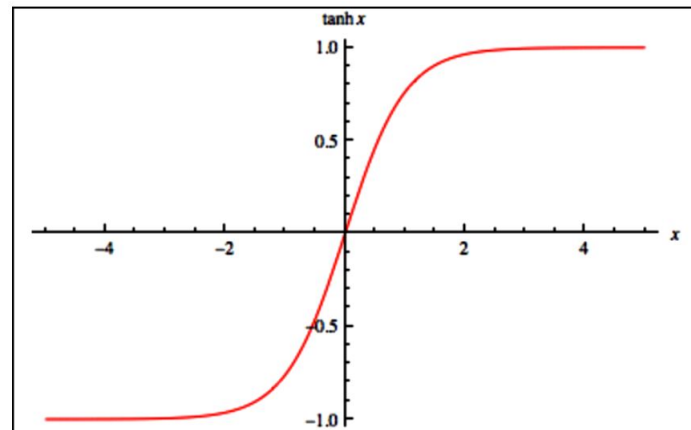


FIGURE 2.3.4: Tanh activation function

**Rectified linear unit (ReLU):** ReLU is a very simple activation function which does not require complex computations, and mostly common activation function used, it outperform the sigmoid and tanh[27]. The formula 2.12 and the curve 2.3.5 shown in following :

$$R(x) = \max(0, x) \quad (2.12)$$

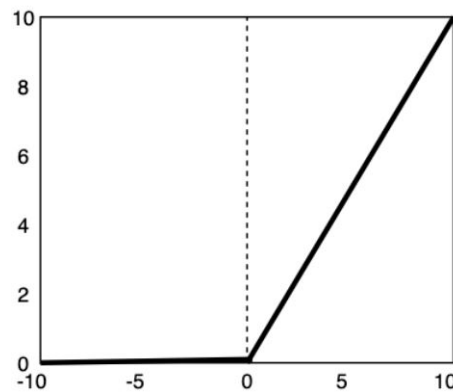


FIGURE 2.3.5: ReLU activation function

### 2.3.3.3 Convolutional Neural Network (CNN)

CNN is a complex and modern architecture for artificial neural network (ANN). The main usage of CNN was for computer vision tasks, analyzing and processing data that has a grid pattern, such as images [48]. The image consist from a pixels, the representation of these pixels by converting the image into matrix with values that indicate the intensity of that pixels. Then CNN applying multiple filters also known as kernels in convolution layer. These filters have a weights that the network update in order to capture patterns and produce a feature map. However, The same CNN architecture applied with NLP to dive in text processing and classification. Figure 2.3.6 show the CNN model dealing with text data.

- **Convolutional layer** This is the first layer in CNN model. The sentence is represented as a matrix of vectors one for each word. For example, the sentence " This method is too complex" is represented by a matrix of five vectors of size 5, i.e. the word 'this' is represented by 5-dimensional vector, word 'method' is represented by a 5-dimensional vector, and so on. This result in a matrix of size 5x5. In the Convolutionl layer, multi filters (kernels) are applied, and an element-wise multiplication is performed and

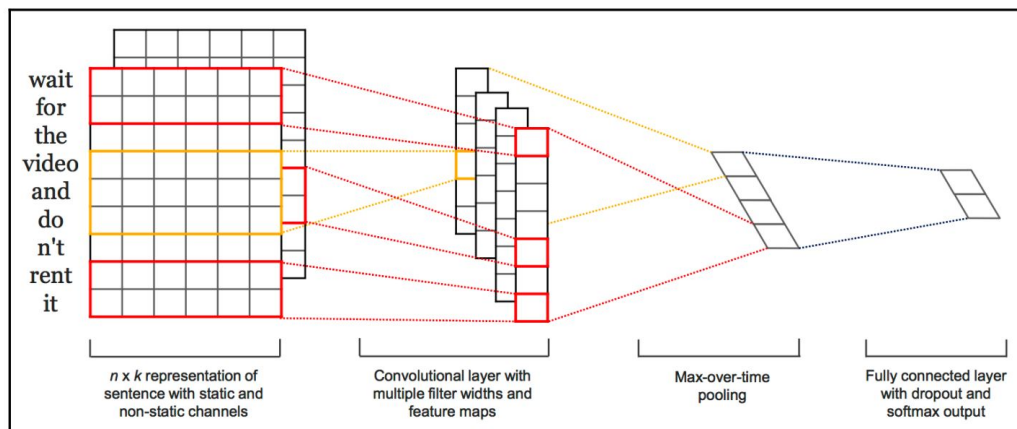


FIGURE 2.3.6: Convolutional Neural Networks for Sentence Classification by Yoon Kim. [32]

then sum the result. See example in figure 2.3.7.

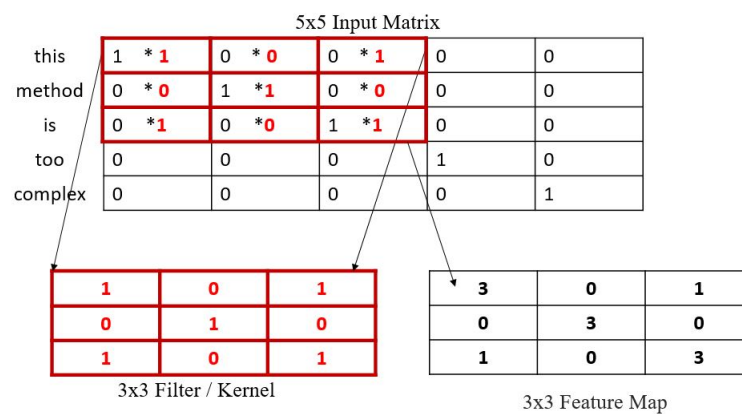


FIGURE 2.3.7: Example convolution operation in 2D using a 3x3 filter. The filter move one step for each operation and add the result in the feature map. then, the filter slide to the right and perform the same operation down.

- **Stride and padding** In certain situations, the size of the feature map will be equal to the size of the input matrix, and this depending on the type of data. Stride is how much the filter move for each step, the default value is 1 as shown in previous example 2.3.7. When stride value increase, the

resulting of feature map get smaller, and some potential locations are ignored. In contrast, padding technique that surrounding the input matrix with zeros to obtains same dimensional for feature map after applying the filter.

- **Pooling layer** Pooling is the next step after the convolution operation. The objective of the pooling layer is to reduce the computation process as minimal as possible, and keep capturing the most important information as much as possible.

There are various types of pooling, to understand these types, the following figure2.3.8 shows the matrix with stride 2 and 2x2 window without overlap as shown in

5	2	6	9
7	1	4	3
3	1	8	2
3	4	5	4

FIGURE 2.3.8: Matrix with stride 2 and 2x2 window

- **Max pooling:** This technique is mostly used in pooling layer, it take the maximum value for each filter window as shown in figure 2.3.9

7	9
4	8

FIGURE 2.3.9: Max pooling

- **Average pooling:** It take the average of 2x2 window.figure2.3.10

4	5
3	5

FIGURE 2.3.10: The averages rounded to the nearest integer.

- **Sum pooling:** This technique just take the sum of the values for each window, as shown in figure. 2.3.11

15	22
11	19

FIGURE 2.3.11: Sum pooling.

- **Fully connected layer** The convolutional and pooling layers extract the features from the input data, the fully connected layer takes these extracted features to get the final classification results. It is the same functionality that happens in ANN fully connected layer.



## Chapter 3

### Literature review

The literature review chapter discusses the metaphor of technical debt meaning in a software engineering context and the definition of self-admitted technical debt, and the extension of term over time by the developers as using the technical debt to workarounds optimal solutions and adopting the sub-optimal. The literature review provides brief information summarized that gather from different resources that include research papers, books, articles, and websites. This information ranges from the appearing the term of technical to the current date in chronological order. The figure shows the spiral method adopted for this thesis in literature writing.

#### **3.1 Technical Debt Metaphor: Definition and Expansion**

Technical debt is a metaphor that was introduced by Cunningham [12] in 1992. Cunningham considers “not quite right code” is technical debt, and explained how the sub-optimal or not complete solution by adopting the short-term would affect the maintainability, which requires more effort in the future, which is what the interest does on incurred debts.

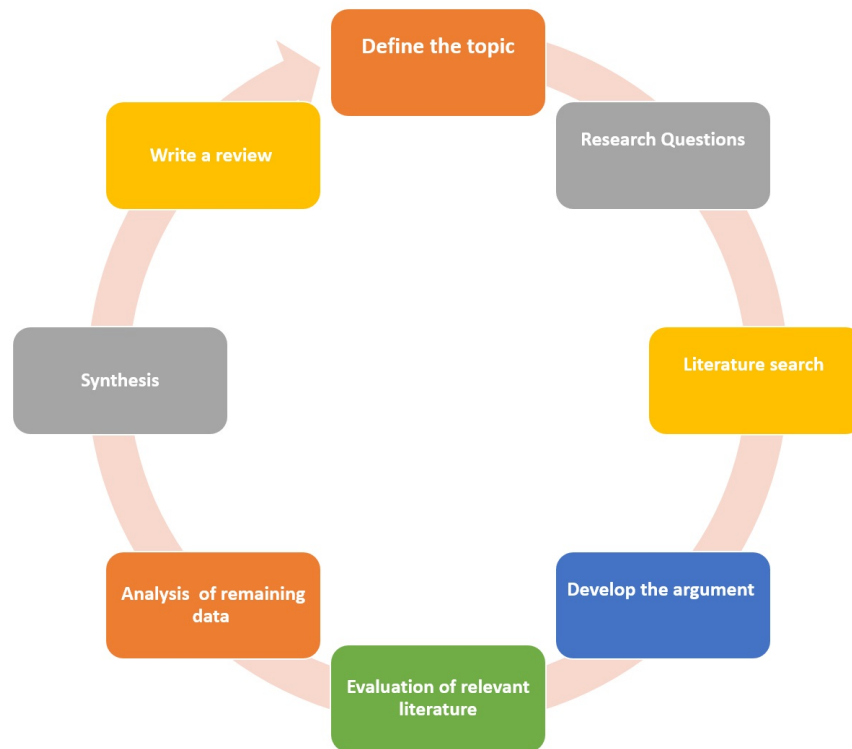


FIGURE 3.0.1: literature review workflow

The term of technical debt continued to refine and expand, Steve McConnell divided the technical debt into two types, intentional debt, and unintentional debt [43]. The first type mostly occurs when the company makes a decision to get the release done and working, without taking into consideration the future, for example, “We don’t have time to build in general support for multiple platforms. We’ll support iOS now and build in support for Android, etc., later.” or “We didn’t have time to write unit tests for all the code we wrote the last 2 months of the project. We’ll write those after we release.”[43]. Moreover, two types of intentional technical debt, short and long term, as with actual debt. Short-term debt is considered on reactively and tactically, commonly as a late stage measure to get a particular release out the door. While long-term debt is strategically incurred , for example “We don’t think we’re going to need to

support a second platform for at least five years, so this release can be built on the assumption that we're supporting only one platform." [43]. This implies that short-term debt must be repaid immediately, maybe as a first version of the next release cycle, while long-term debt can be repaid for a few years or longer.

The unintentional debt is the non-strategic result of the bad implementation; this debt commonly occurs with insufficient knowledge and experience, that range between design to a junior developer who writes smell code unintentionally [43].

Otherwise, Martin Fowler [21] introduced four quadrants types of technical debt, reckless, prudent, deliberate and inadvertent.

**Reckless inadvertent debt :** The team in this situation there has no choice about incurring the debt (eg. not aware of design practices), either does not know it or if the debt exists the team can not correct it, so the team should stop development and develop itself, to know the technical debt types and effect to allow them to make a rational decision about whether or not it is wise to continue with.

**Reckless Deliberate debt :** This kind of debt occurs with previous knowledge that the team incurring the debt, but they choose to go "quick and dirty" instead of writing clean code because they assume that they can not afford the time to do so.

**Prudent Deliberate debt :** In this case shortcuts quality made by the team consciously because of an imminent deadline. The team knows about the debt and its consequences, and commonly adds its correction to the backlog, but this time the focus is on the delivery rather than quality. Prudent inadvertent technical debt : This case represents the over quality of the project, where the team has the skill to applying the best design and development, that satisfying the requirements, but over time there becomes a difference between what has been

built and what is required, because of the deep in understanding of the system, and this lead to falls into the technical debt within the category of prudent

Figure 3.1.1 presents the technical debt quadrant, each quadrant illustrates the situation that can arise while a team is working on a software project.

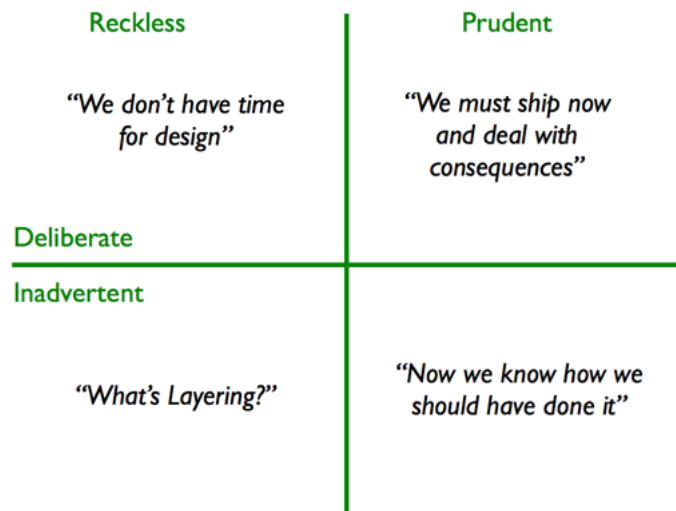


FIGURE 3.1.1: Technical Debt Quadrant [21]

Alves et al [3].Proposed an initial taxonomy of technical debt types and strategies to identify and manage TD. In this study, the focus will be on the five types of TD and their definition. Nicolli found in her systematic mapping study the following types of debt in the literature, and he provided the definition of this kind

- **Design debt:** The violation of the design principle of good object-oriented design, it can be found through analyzing the source code.
- **Documentation debt:** When the software has a problem in documentation such as inconvenient, or incomplete of any type of documenting.
- **Test debt:** The issues that affect the software quality and related to the tests. Example of this type, lack of test coverage.

- **Defect debt:** Indicates to the known defects and defined by the users as a bug or by the team itself, and it has been agreed that should be fixed, but since the limited resources and other priorities , have to be delayed until later.
- **Requirements debt:** It indicates the trade-offs made regarding what requirements the team necessarily to implement or how they are implemented. Such as partially implemented requirements, fully implemented but not satisfied all non-functional requirements.

Additional types of debt include code debt, service debt, infrastructure debt, usability debt, people debt, process debt, build debt, test automation debt, and **Architecture debt:** that refer to problems faced in system architecture, such as violation of modularity that may be effective on the quality attributes (performance , robustness , etc..). This kind of debt is not paid easily, because the fix is not at the level of refactoring the code, but rather more extensive development activities.

### 3.2 Identification of Technical Debt through code-base

This study will focus on TD identification. That is the first step in technical debt management forward to removing this debt. More strategies to identify TD, either human-based such as source code analysis or automation static analysis, include two types of code smells and issues. Code smells can be identified as design debt. Fowler and Beck [22] first introduce the bad smells (code smells) term and the code smell described as the pattern that is less than ideal and violates the rule of object-oriented design and should be refactored. The authors

suggested continuous refactoring by reviewing the code and identifying code smell during development.

Yamashita and Munin [67] studied the interaction between 12 code smells and analyzed the relations between this action and how related to the maintenance of the project. The investigators found empirical proof proved that inter-smell relationships were correlated with problems during maintenance. Soltanifar et al. [60] In order to generate defect prediction models, data science and analytics techniques were investigated. The consequence was that code smells are a strong indicator of defect proneness in the software. Furthermore, this literature indicates that at least an associative association between code smells and maintenance and the connection between TD and maintainability has been identified, which lead to the use of code smell detection as an approach to TD identification.

Other kinds of identify TD are commonly used in literature automated static analysis, that extraction information about software from its source code using automatic tools[8]. These tools search for the issues that violate programming practices, that might cause faults, and negatively impact software quality (e.g., performance, maintainability). The automated static analysis issues at the source code line level are generally more accurate than code smells at the class or method level. That study indicates little overlap between issues and code smells methods when used to identify technical debt.[70] Issues should be eliminated by refactoring to prevent problems that might arise in the future and accumulate these issues constitute technical debt. Sonarqube (<https://www.sonarqube.org/>) is one of the popular open-source automated code-based static analysis tools, both adopted in academia [33] and in industry [61]. It supports more than 27 languages and is used by more than 85K organizations, and supports CI/CD

integration. It is important to note that the concept of code smells used in sonarqube does not refer exactly to code smells defined by Fowler et al.[22]Sonarqube rules for classified the code smells when the violation of ( long method, large class, duplicated code, and long of parameters ). [34].

### **3.3 Identification of Technical Debt through Source Code Comments (Self-Admitted Technical Debt)**

The previous section discussed the approaches that used to detect technical debt, whether human-based analysis that manually inspection in the source code to find the poor code that violate the recommended practices development principle and form it in technical debt, or automated static analysis code-based technique, that in most case adopting rules to detriment the code smells and issues, the Sonarqube is an example tool as mentioned above. Identification and management of technical debt are considered an open challenge. On the other hand, Potdar and Shihab [53] were the first to analyze the comments in the source code to identify the technical debt, and introduced the concept of “Self-Admitted Technical Debt”. Unlike common static analysis code-based tools that depend on predefined rules, metrics, and thresholds to expect debt, technical debt refers to the code defective, incomplete, smell, or temporary, and intentionally written by the developers (self admitted) with obvious recognition that implementation is not optimal. These developers admitted by themselves that piece of code is technical debt documented through comments. The authors explore source code comments in four open-source projects to study the amount of SATD used in these projects. They also investigated why the developers used this debt in the projects and how the SATD was removed from the projects. The result was SATD exists in 0.024 - 0.31 of the files, most of the SATD was introduced by the

developers with higher experience, and there is no relation between SATD and time constraints and code complexity. Finally, 0.263 - 0.635 of SATD is removed from projects after being presented. Moreover, in this study, Potdar and Shihab introduced 62 patterns to indicate the SATD, through manually reading 101,762 comments to define patterns that indicate SATD.[53]

This approach allows detecting the SADT simpler than manual inspection of the comments, but the number of projects used may be limited to generalization to detect SATD in other software systems. Additionally, the technical types are not defined in this study.

Consequences on previous work , Maldonado and Shihab [41], used five open-source projects and manually inspected and read through source code comments of 33K comments, to explore the type of SATD, the result was introduced five types of technical debt, requirement, design, defect, test, and documentation debt, most of them was design debt, and dataset includes comments classified to these five kinds of technical debt. The authors adopted the measure of classification as follows:

- **Self-admitted design debt** : Comments that include problems with the design of the code. The comments that indicate lack of abstraction, long methods, bad implementation, misplaced code and workaround or temporary. For example *"TODO: - This method is too complex, lets break it up"* - [from ArgoUml] *"/\* TODO: really should be a separate class \*/"* - [from ArgoUml]
- **Self-admitted defect debt** : The comments describe the part of code that has a problem in expected behavior, in other words it has a defect in code. *" // Bug in above method"* - [from Apache Jmeter]" *"// WARNING: the OutputStream version of this doesn't work!"* - [from ArgoUml]



- **Self-admitted documentation debt** : The expression of comments that lack of documentation describe the part of software. *“\*\*FIXME\*\* This function needs documentation” - [from Columba]*
- **Self-admitted requirement debt** : Incompleteness of program, class or method. Additionally, the requirement which are implemented, but does not absolutely satisfy all the non-functional requirements (e.g. performance ,security, etc.).[4]  
*“/TODO no methods yet for getClassname” - [from Apache Ant]*
- **Self-admitted test debt** : The comments that express need to improve current tests or implement new tests . *“// TODO - need a lot more tests” - [from Apache Jmeter]*

The authors in this study depended on the manual classification, by reading a large amount of comments, the first author who classified all comments that take 95 hours [41] may lead to bias, but the definitions of the types self admitted technical debt and supporting by example as appear above are convincing.

Freitas Farias et al. [23], Introduced the Contextualized Vocabulary Model CVM-TD, the aim of the study was to identify the different types of SATD in comments of source code, the CVM-TD model depends on identifying the classes: nouns, verbs, adverbs, and adjectives, that used by developers such as “TODO, FIXME”, and related to software engineering concepts, the goal of applying model is to identify the contextualized structure of terms for supporting the detection of different types of TD through comment analysis relies on TD vocabulary provided by the model. The evaluation of the model showed that comments that returned by the model were different from the comments which were evaluated to contain the SATD. This results in low detection performance and

needs to enhance how classes of words are mapped to various types of SATD to enhance the model.

Bavota and Russo [7] introduced differentiated replication of the work by Potdar and Shihab in [53] with large scale. The study was run on 195 software projects, with 600K commits and 2 Billion comments , to investigate the spread and evolution of SATD, and the relation between these debts and software quality. The main results showed that SATD are distributed in an average of 51 instances for each system. Moreover, even when fixed this debt, it survives a long time on average more than 1000 commits. Additionally, there are other studies examining the relationship between the SATD and quality of software. Wehaibi et al. have been investigating whether the files that include SATD have more chance to include defects in comparison to the files that not include SATD, and the changes in the SATD introduce defects in the future. The result of study presented that the self admitted technical debt in addition to the negative impact on the system, it related to defects, and it making the change more complicated in the future.[63]

From another side, Maldonado, Everton da S. et al [42]. Studied five open source projects to examine who removed SATD, how much removed from and how long SATD live in the project. The authors extract the comments from the project using open source library SrcML, traced the comments in all versions of the system for each project, and adopted the natural language processing (NLP) based technique, that proposed by Maldonado et al.[59] to classify the comments to technical debt. The empirical result for this study was the majority of SATD comments are removed on average 0.749, most of SATD average 0.544 removed by the developer itself who produced it, the life of SATD in the project was on average 18.2–172.8 days.

The work mentioned so far gave a general idea about technical debt and self

admitted, and how the researchers investigate different fields starting from identification, management, evaluation, and removal. Next, the focus will be more on the study goals and deep in recent literature related to the identification of self-admitted technical debt. For any purpose to the treatment of self-admitted technical debt, the identification is imperative as the first step, especially when dealing with a large number of comments that need to classify to the technical debt or not. After that if there way to add more details to this classification to know the types of this debt, when the reason of debt known such as there is a problem in the requirements, the cause will know. Then, the problem can handle quickly with less interest. Therefore, the focus will on the detection category related to this study. Three areas of research found in the literature include pattern-based approaches that depend on identifying textual patterns in comments. The machine-learning approach depends on the automation techniques such as NLP, machine learning classifiers. Finally, Deep-learning approaches this based on more advanced techniques such as a neural network.

### **3.3.1 Pattern-based approaches**

Reference to what discussed above Freitas Farias et al. [25], more investigations are added to the previous work that introduced “Contextualized Vocabulary Model” for identifying TD (CVM-TD) by Freitas Farias et al. [23]. The authors extend the research [23] to characterize the factors that influence the accuracy of overall TD identification by additional study and participate software engineering master students with different English skills and development experience. The comments that generated from the CVM-TD model given to the three expert researchers to create an oracle of comments identify as technical debt, and 36 Software Engineers with different experience and English language skill level

to flag those suggesting of TD introduced from the model , 4 participants eliminated from the experiment due to not complete all data needed. The experiment result showed that the skill reading affects the identification TD, but the classification is not affected by the experience. On the basis of that, the CVM-TD model's output served the developers whether they had experience or not, the average accuracy of 0.673 when TD comments are detected, which outperform previous work;[23]. Note in this experiment that it is still in the same scope of the previous study, and it depends on the pattern of words in the comments (e.g. "most memory consuming", "memory allocated") may refer to build issues, without considering the meaning and sentiment for the whole comment.

Freitas Farias et al.[24] Proposed the work that applies, evaluates, and improves previous work of contextualized patterns to detect SATD using source code comments that analysis in the studies [23][25]. Three empirical studies were performed to do that, 23 participants analyze pre-defined contextual vocabulary patterns and score their level of importance in identifying SATD elements. A qualitative empirical study conducted for analysis and examine the relationship between each type of debt and pattern. Finally, a feasibility study was conducted using a new vocabulary, which was improved based on the results of previous empirical studies, to automatically identify self-admitted technical debt and the types of debt found in three open-source projects. More than 0.50 of the new patterns are critical to technical debt detection. The new vocabulary was succeeded in finding items related to code, design, defect, documentation, and requirement debt. The main contribution of this study was the identification of self-admitted technical debt depending on the knowledge embedded in vocabulary, which can be used to automatically identify and classify TD by analyzing the source code comments.

### 3.3.2 Machine-learning approaches

Maldonado et al. [59] used NLP maximum entropy classifier (Stanford Classifier) approach to automatically identify SATD from the comments, including design and requirement TD. The authors used 10 open source projects, extracted 62,566 comments, and classified them manually to create a dataset with five types of TD: requirement, design, defect, documentation, and test debt. The experiment used 10 fold cross-project validation, 9 open-source projects for training, and one project for validation. The result presented that the NLP improved the identification accuracy compared with previous pattern-based detection. The classifier scored an average F1-measure of 0.620 for design debt, 0.403 for requirement debt, and 0.636 technical debt without types. Additionally, the study also provided top-10 lists of textual features that the developers used as SATD, that means there is a variety of style of expression of the SATD. To obtain a satisfactory size of comments for training stage. The dataset included 3,900 comments, and only 195 comments are necessary to be a design debt. For requirement debt 2,600 comments and 52 comments classified as requirement TD, it is possible to get high accuracy. So, using just 0.05 of the comments as a requirement and 0.09 as design for training, the best performance was 0.80. Moreover, when used 0.23 as training for both requirement and design, the best performance was 0.90.

Flisar and Podgorelec [19]. Investigate in word embedding method and enhance feature selection to identify SATD. In this study, more than a million unlabeled comments extracted from 360 open-source projects were used. These comments have been used to build the word2vec model in semantic space. Then the created model is used to detect similar features in comments. Three steps used to enhance the feature selection, top k% of feature, cosine similarity between the

feature  $X$  and all other features, and each feature word that are most semantically similar are selected and added to the feature list. For evaluation of the experiment, the dataset used in [59] that includes 3,298 labeled comments, the results achieved 0.82 of correct predictions of SATD. The authors expanded the study to identification that comments belong into two classes whether SATD is included or not, [20]. In this study, the same dataset in the previous study was used, and it proposed a feature-enhancement approach and used three feature selection methods (CHI, IG, and MI), with three text classifiers algorithms (NB, SVM, and ME). The main contribution of this paper was to build the word embedding model that detects the semantic meaning in related words in the comments using the Word2vec model. The feature selection method proposed, by combining three methods of feature selection (CHI, IG, and MI) used in the next step of learning algorithms to enhance classification results. The prediction model achieved 0.82 of correct predictions of SATD.

Other studies adopted the text mining approach to identify and manage technical debt. Huang et al. [28] introduced an automated approach to detect SATD using text mining, the comments extracted from 8 open-source projects, and used as training to predict the type of the comment in a new target project. All comments have been pre-processed to extract features through applying stop-word removal, tokenization, and stemming techniques. Vector Space Model (VSM) is used to represent the comment as a vector. In VSM, a feature is viewed as a dimension, and a comment is represented as a data point in a high-dimensional space. To mitigate the curse-of-dimensionality problem that faced the study, a feature selection, namely Information Gain (IG), was applied to select the top 0.10 useful features. The features selected to use as a train sub-classified on each source project, for predict the label of comment for target project, these sub-classifiers are trained using a Naive Bayes Multinomial (NBM) approach to

define the label of a comment depending on the number of involving features. A combined classifier takes the vote for each comment per sub-classified to predict the final classification ( i.e. 5 sub-classifier predict SATD, and 4 predict not, the final classifier result is SATD). The results of the experiment was, for each target project, the best performance achieved in terms of F1-score, with an average of 0.737. In the same context, SATD detector tool proposed to automatically detect SATD using text mining-based [37], its implementation for the previous work [28]. This tool can be used as an Eclipse plug-in to identify SATD on Java code, the back-end of this tool is a pre-trained composite classifier to detect SATD comments. This tool will be used in this study to extract the comments from open-source Android applications. Wattana Kriengkrai, Supatsara, et al [62]. Introduced an approach using N-gram IDF, with multi classification techniques, and built a model that can identify a comment to design debt, requirements debt, or non-SATD .Maldonado et al. [59] dataset that contains 62K Java source code comments used in this study. For mitigating the imbalanced dataset, Instance Hardness Threshold (IHT) approach used by determined threshold value to remove often useless samples that are misclassified, Random Forest (RF) machine learning algorithm applied to classify target comments. The result of the experiment was N-gram IDF outperform traditional techniques BOW and TF-IDF, with the average F1-score values of 0.6474.

Zhe Yu et al. [69] Proposed Jitterbug framework with two methods for identifying SATD. The dataset proposed by Maldonado et al. [59] used. The first type, "easy to find", that can be detected automatically without human intervention, because the comment has explicitly denoted that include the keywords or pattern that relate to type of SATD such as "Todo", "Fixme". This approach can find 0.20-0.90 of SATD automatically. The precision of identifying SATD close to 0.100, when using a pattern recognition technique. On average, those comments

cover 0.53 of the total SATD. Second type: “hard to find” is not easy to classify and needs experts to accurately decide, and only humans can make the final decisions, and it is still hard for algorithms. In this approach, supervised machine learning is used to present the comment for the experts to identify SATD that is not identified automatically. After that, the comments identified by the experts reuse by update the model and continuous training.

### 3.3.3 Deep-learning approaches

Ren, Xiaoxue, et al [55]. It has proposed a Convolutional Neural Network (CNN) technique to classify the comments to SATD or non-SATD. This approach included four stages. First, in the training model, the source code comments are used as an input with the corresponding label classified as SATD or not SATD, then this model is used in the prediction stage to predict if the comment SATD or not with given unseen label comments. Moreover, the trained model is deconvolutional in keyphrase extraction to extract main phrases from the input comments that lead to classification decisions into a set of intuitive SATD patterns. The result of this study was, on average F1-score of 0.752 within-project. The result for cross-project prediction indicates that the CNN model more generalizable than the text mining approach with limited training data. The CNN learned model could derive SATD patterns more comprehensive than the 62 patterns identified in the study of Potdar and Shihab [53]. The contribution of this study, substantial enhance text mining method within project and cross project, by using CNN to identification the SATD from source code comments, backtracking method designed to extract key phrases and the patterns of SATD from the source code comments, which enhance the SATD classification results with CNN model. Comprehensive experiments were conducted to evaluate the



performance and to generalizability and adaptability. Furthermore, the explainability of the CNN learned model from SATD features and patterns.

Santos, Rafael Meneses, et al [57]. They evaluated a neural network Long short-term memory (LSTM) model with Word2vec, for identification requirements and design SATD in the file of source code comments. The evaluation was done by experiment using the dataset for 10 projects collected and labeled by Maldonado and Shihab. The dataset was divided into two groups; the first group contains 60,907 comments, and 2703 are design debt. The second group includes 58,961 comments, 757 from them are requirement SATD, and the other comments for tow groups without SATD. The independent variables were LSTM , Word2vec model, and the dataset. Dependent variables were model prediction represented by precision, recall, and f-measure. The results of classification using the leave-one-out cross-project validation process showed that:

- Without pre-processing the comments, and using Word2vec , the LSMT achieved higher recall than Autosklearn, and Maximum Entropy , but loses in precision and f-measure.
- With pre-processing the comments, and applying Word2vec, it had improvement in precision, and the f-measure.
- Without Word2vec , the result was 0.56 improvement in recall in the design SATD classification.
- With Word2vec, the improvement in recall was approximately 0.36 in both SATD types.

The summary of results for this study was: The LSTM model without Word2vec achieved greater recall, but performed worse in precision and f-measure.

The same authors have another publication [56] that presents the same previous study, but the word embedding (Word2vec) excluded from this study, and

the result was : compared with two NLP approaches: auto-sklearn and maximum entropy classifiers. Average precision was improved by around 0.08 compared to auto-sklearn and 0.19 compared to maximum entropy, however, the LSTM model had worse results in recall and f-measure.

### **3.4 Identification of Self-Admitted Technical Debt Using Commits Messages**

Most of the studies that noticed during literature were focused on the source code comments more than commits to the identification of SATD. From the studies that were used the commits approach to dealing with SATD, Bavota and Russo [7] that mentioned in section 3.3 , and of the results, the number of SATD comments are growing over time, and it takes more than 1000 commits on average before SATD is removed or fixed. Yan et al [68] Introduced change-level self-admitted technical debt determination. This model determines whether the change introduces SATD, by using all versions of the source code comments and identifying the SATD at file level for each version. After that, manually label the changes that introduce SATD, and extract 25 features referring to three dimensions, diffusion, history, and message. In the message dimension, they viewed commit messages that written by the developer and analyzed to extract information about the purpose of change in the code. This information grouped into five categories ("has bug, has feature, has improve, has document, has refactor"),the commits that contained the keyword will belong to one of five category ( i.e. "bug" indicates fixing bug) , the commit length added to predict if the commit is related to SATD depending on the number of keywords that are included in the message. The random forest. Across 7 projects with 100,011 changes, this

model achieves an AUC of 0.82, and a cost-effectiveness of 0.80 for all features. For messages commit achieves AUC of 0.57, and a cost-effectiveness of 0.72.

Maipradit, R. et al. [40] introduced on-hold self admitted technical debt, that performed a qualitative study on 333 commits extracted from Maldonado, Everton da S., et al [42] study that labeled the commits as removed, that means the commits indicate to remove related technical debt. On-hold refers to debt that has a condition that makes the developer waiting for another event or action, such as adding functionality elsewhere to fix the debt, the aim of the study to identify these on-hold instance debt, and detect the specific conditions that make the developer waiting. The study achieved an area under (AUC) of 0.98 for the identification, and 0.90 of the specific conditions are detected correctly.

Rantala & Mäntylä [54]. Replicating and extending the work introduced by Yan et al.[68], they used 1876 commits messages extracted from five repositories (Camel, Log4J, Hadoop, Gerrit, and Tomcat) that pre-labeled as SATD, and three techniques of NLP (bag-of-words, latent dirichlet allocation, and word embedding), to predict self-admitted technical debt from commit messages. The main contribution of this study, the bag-of-words technique is the best performance with median (AUC 0.7411). Automatic feature selection from the commit message improved the prediction performance for SATD. For generalization results when using different repositories, the words in the commit message are required to appear in several repositories, with a certain number of times. List of words that can be used to predict the SATD in commit message contents.

The summary of the previous studies that related to this approach appear in the table 3.1

Paper	Approch	Main contribution(s) / Finding(s)
[53] (2014)	Pattern-based	SATD exists in 0.024 -0.31 of the source code files. 0.263 - 0.635 of SATD gets removed from projects after introduction. Introduced 62 patterns to indicate the SATD
[41] (2015)	Filtering heuristics	Dataset classified to five types of SATD (requirement, defect, design, documentation, and test)
[23] (2015)	CVM detection	Provides a vocabulary that may be used to detect TD items.
[25] (2016)	Pattern-based	Set of Patterns to identify TD in comments.
[59] (2017)	NLP and Max Entropy classifier	Data set of classified SATD. Achieve an average F1- measure of 0.620 when identifying design self admitted technical debt, and an average F1-measure of 0.403 for requirement TD.
[28] (2018)	Text-mining , ML classifiers (NB, SVM, KNN,)	F1-score, with an average of 0.737 for requirement and design TD.
[19] (2018)	Word2vec, feature selection , SVM	Created Word2Vec model from 360 open source java projects. 0.82 of correct predictions of SATD or not.
[20] (2019)	word2vec, feature selection(CHI, IG and MI) . classification algorithms (NB, SVM, and ME)	0.82 of correct predictions of SATD or not.
[62] (2019)	RF classifier, N-gram IDF	Multi classification techniques, and construct a model that can identify a comment to design debt, requirements debt, or non-SATD. F1-score, with an average of 0.648
[55] (2019)	CNN	Backtracking method designed to extract key phrases and the patterns to using in CNN.The result of the study was on average F1-score 0.752 for prediction SATD or not within project.
[57] (2020)	LSTM , Word2vec	LSTM model with Word2vec have improved in recall and f-measure. The LSTM model without word embedding achieves greater recall, but perform worse in precision and f-measure

TABLE 3.1: Summary papers that related to this thesis

# Chapter 4

## Research Methodology

The main objective of this thesis is to classify the SATD comment into what category that belongs to (requirement, defect, design, test, or documentation), by using comments and commits written by the developers. NLP techniques are used to extract the features from these comments, then machine learning and deep learning are used as classifiers. The figure 4.0.1 shows the workflow of the research methodology.

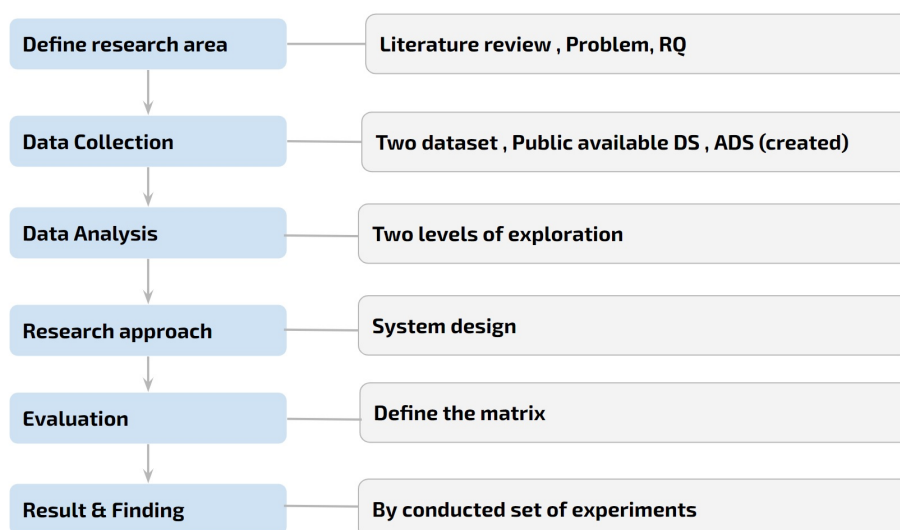


FIGURE 4.0.1: Research methodology workflow

## 4.1 Dataset description

To perform this study, more than one dataset will be used. The developers used two types of expressions for writing, which can be considered as SATD; source code comments and commit messages. The focus will be more on the comments and the commits used to add more variety of sentences used in Adataset that will be created, with the aim of generalizability.

### 4.1.1 Source code comments

The first dataset that usually used in most of previous studies and introduced by Maldonado et al. [59] the data set consist of 62k comments that extracted from 10 open source project ( Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby and Squirrel SQL ), the comments classified into five types of SATD, requirement : 757, design : 2703, defect: 472, test: 85, documentation: 54, other comments are unclassified. Figure4.1.1 show the sample of data.

apache-ant-1.7.0	DEFECT	//FIXME: Is "No Namespace is Empty Namespace" really OK?
apache-ant-1.7.0	DEFECT	//FIXME this is actually not very cpu cycles friendly as we are converting from // dos to java while the underlying Sun implementation will conver
apache-ant-1.7.0	DEFECT	// check first that it is not already loaded otherwise // consecutive runs seems to end into an OutOfMemoryError // or it fails when there is a nat
apache-ant-1.7.0	DEFECT	// TODO: // A warning line, that shows code, which contains a variable // error will cause some trouble. The parser should definitely // be much b
apache-ant-1.7.0	DEFECT	// todo: is this comment still relevant ?? //FIXME: need to use a SAXSource as the source for the transform // so we can plug in our own entity re
apache-ant-1.7.0	DESIGN	/** XXX I really don't like this - the XML processor is still
apache-ant-1.7.0	DESIGN	//JUnit 4 wraps solo tests this way. We can extract // the original test name with a little hack.
apache-ant-1.7.0	DESIGN	//XXX ignore attributes in a different NS ( maybe store them ? )
apache-ant-1.7.0	DESIGN	//borrowed from TokenFilterTest
apache-ant-1.7.0	DESIGN	//XXX Move to Project ( so it is shared by all helpers )
apache-ant-1.7.0	DESIGN	// What is the property supposed to be?
apache-ant-1.7.0	DESIGN	// PR: Hack for ant-type value // an ant-type is a component name which can // be namespaced, need to extract the name // and convert from c
apache-ant-1.7.0	DESIGN	// there's a convenient xsלטc class version but data are // private so use package information
apache-ant-1.7.0	DESIGN	// stolen from org.apache.xerces.impl.XMLEntityManager#getUserDir() // of the Xerces-J team // which ASCII characters need to be escaped
apache-ant-1.7.0	DESIGN	// bingo append it. (this should normally not be done here)
apache-ant-1.7.0	DESIGN	// This is faintly ridiculous:
apache-ant-1.7.0	DESIGN	//cannot remove underscores due to protected visibility >{

FIGURE 4.1.1: Sample of comments dataset [59]

To create a new dataset, so 222 source code comments extracted from two open source android mobile applications, namely K9 and wordpress, using a

Adataset		
Project Name	No. Comments	Removed duplicate comments
Gerrit	272	172
Camel	4332	1162
Log4j	136	91
Tomcat	1318	1052
Hadoop	1165	625
K9 App	339	145
WordPress App	94	77
<b>Total</b>	<b>7317</b>	<b>3324</b>

TABLE 4.1: Number of new collected comments

SATD detector that automatically detects SATD using text mining-based algorithms [37]. In K9 app, there are 339 comments extracted by this tool, 170 comments as SATD and 169 as Java tasks, after removing the duplicated comments 145 comments used that classified as SATD . In the Wordpress app 94 comments are extracted as SATD, and reduced to 77 after removing the duplicate comments. Additionally, 3102 comments were took from five open source code projects(Gerrit, Camel, log4j, Tomcat, Hadoop ) that were used by Everton da S. et al. study [42], and the comments classified as SATD. These comments will be manually labeled into the types of SATD that include requirement, design, defect, documentation, and test.The table 4.1 summary the final comments.

#### 4.1.2 Commits messages

For the commits messages the detest [54] will be used and it requested from the author via email. The dataset contains the 73,625 messages from which 1,876 classified as SATD. The commits messages are in their original state, without any processing steps. After remove duplicate commits, and URL of conduit for changesets between subversion and Git for most of the commits, the dataset

have 1758 commits classified as SATD. Figure 4.1.2 show the sample of commits from dataset.

ProviderException is not needed.	0	camel
Exception hierarchy is being revised. Invalid[URIEndpoint/URL]Exception	0	camel
Fixing Logging level.	0	camel
Unicode number is now used for some of the characters.	0	camel
TODO: GSM 0338 pattern needs to be checked. It is used as an	1	camel
Fix some checkstyle issues.	0	camel
Custom InvalidPayloadException is removed. Camel 's	0	camel
LOG instance removed. Parent Logger is going to be used.	0	camel
Fixing. This component info logging should be TRACE or DEBUG. Now, it	0	camel
Fixing. Logging and throw. Do not log and throw error, just throw error.	0	camel
Fixing lack of test scope camel-spring-javaconfig.	0	camel
camel-cm component added following the steps in	1	camel
	0	camel

FIGURE 4.1.2: Sample of commits dataset; 1 classified as SATD, 0 unclassified [54]

### 4.1.3 Manual annotation

After preparing the comments from source code of seven software projects and the commits from previous study [54] as aforementioned. The manual annotation was performed through two phases. In phase 1; the classification method from Li, Yikun et al study [35], and Ai Deng study [15] will be followed, which both are based on a framework proposed by Alves et al [4]. Additionally, Alves et al [3] conducted systematic mapping in total, 100 studies, dated from 2010 to 2014. The taxonomy and definitions at least for the five types of TD used in this study was the same.

Alves et al [4] proposed an ontology to definitions and indicators of technical debt that were spread across the literature. In other words, the factor that lead to the introduce a technical debt. Alves et al provide 13 different types of TD with definitions that include: "architecture, build, code, defect, design, documentation, infrastructure, people, process, requirement, service, test automation, and



test debt". In case of self admitted technical debt, Maldonado and Shihab[41] manually analyzed 33,093 comments and classified them, the main finding was; most of technical debt types that self admitted in source code are (requirement, design, defect, test and documentation debt). The other 8 types that remaining of technical debt defined by Alves et al. [4] were not found in this approach since the developers are not likely express them by the comments, i.e infrastructure, people and process no indication was found. Additionally, some of TD may overlap such as design and architecture.

To create the dataset, and after prepared the comments and commits, the database created and all the comments and commits inserted for classification. Then, and after the comments classified in phase 1. Public website <sup>1</sup> was published to git the maximum number of classification from the participant. The website include three pages(Home,Registration, Classification). In the home page the steps provided for participant to complete the task, and set of definitions to deep understand the technical debt and their types as following :

- **Source code comments** : Source code comments are explanation or annotation that written by the developers, comments allow developers to clarify, document, and express concerns about the implementation in an informal method that does not influence the functionality of the program, and are generally ignored by compilers and interpreters [65].
- **Commits**: commits messages are an express of action that the developers made on the source code and document this action with semantic commits.
- **Technical debt**: is a metaphor, coined by Ward Cunningham [12]. It reflects the additional cost that imply to rework caused by a sup-optimal solution instead of using the better approach in software development life

---

<sup>1</sup><https://github.com/asabbah44/SATD>

cycle. The concept of TD is derived from financial debt, as the Interest resulting from the late payment. Similar to the financial dept, TD has an interest and the cost increases if not pay the debt early, by refactoring the code on the suitable time, to avoid interest in the future.

- **Self-Admitted technical debt - SATD** : It is a technical debt that written by the developers deliberately, through comments or commits messages, with the knowledge that the implementation is not an optimal solution for the software [28].

The definitions for each type of self- admitted technical debt provided obviously, in order to make the reader understand the various types of SATD, and given examples for each type as following:

- **Self-admitted design debt**: The comments that indicate a problem with the design of the code. It could be comments about , lack of abstraction, long class and methods, bad of implementation, misplaced code, workarounds, or a temporary solution.[3]. The following source code comments are examples of self-admitted design debt:
  - // PR: I do not know what to do if the object class // has multiple defines // but this is for logging only... -[from apache-ant-1.7.0]
  - // I hate this so much even before I start writing it. // Re-initializing a global in a place where no-one will see it just // feels wrong. Oh well, here goes." - [from ArgoUml]
  - TODO: - This method is too complex, lets break it up - [from ArgoUml]
  - TODO: really should be a separate class - [from ArgoUml]

- TODO: move this to components – the only reason why it’s in core is because // it’s used as a guinea pig by a couple of tests.-[from apache-jmeter-2.10]
- // XXXX re-evaluate this //can getSuper work by itself now? //If we’re a class instance and the parent is also a class instance //then super means our parent.-[fromjEdit-4.2 ]
- **Self-admitted defect debt:**The comments describe the part of code that has a problem in expected behavior, in other words it has a defect in code. [41].
  - “// Bug in above method” - [from Apache Jmeter]
  - “// WARNING: the OutputStream version of this doesn’t work!” - [from ArgoUml]
  - // FIXME formatters are not thread-safe-[from apache-ant-1.7.0]
  - // TODO: Something might go wrong during processing. We don’t really // want to create the model element until the user releases the mouse // in the place expected.[from-argouml]
  - // todo: is this comment still relevant ?? // FIXME: need to use a SAXSource as the source for the transform // so we can plug in our own entity resolver-[from apache-ant-1.7.0]
- **Self-admitted test debt:** The comments that express need to improve current tests or implement new tests. "Inadequate test coverage, lack of tests, and improper test design" [41].
  - // TODO should this be done even if not a full test plan? // and what if load fails?-[from apache-jmeter-2.10]

- // not sure whether this test is needed but cost nothing to put. // hope it will be reviewed by anybody competent-[from apache-ant-1.7.0]
  - // cleanAllExtentsBut(model); // TODO: why is this causing a crash?!?- [from ArgoUml]
  - //TODO: Test Mac keyboard accelerator changes done here by mlivingstone // shortcut key
- **Self-admitted requirement debt:** Comments indicate that there is an ambiguous requirement that leads to incompleteness of program, class or method.[41]. Additionally, the requirement which are implemented, but does not absolutely satisfy all the non-functional requirements (e.g. performance ,security, etc.).[4]
    - “/TODO no methods yet for getClassname” - [from Apache Ant]
    - “//TODO no method for newInstance using a reverse-classloader” - [from Apache Ant]
    - “TODO: The copy function is not yet \* completely implemented - so we will \* have some exceptions here and there.\*//” - [from ArgoUml]
    - // TODO support multiple signers -[from apache-jmeter-2.10]
    - // Set the overall status for the transaction sample // TODO: improve, e.g. by adding counts to the SampleResult class-[from apache-jmeter-2.10]
  - **Self-admitted documentation debt:** incomplete comments, lack of code comments, no documentation for important concerns, poor documentation. The expression of comments that lack of documentation describe the part of software.[41]

- \***FIXME**\* This function needs documentation-[from columba-1.4-src]
- // **FIXME**: Document difference between warn and warning (or rename one better)-[from jruby-1.4.0]
- “// **TODO** Document the reason for this” - [from Apache Jmeter]

Before starting the manual annotation process, each participant needs to fill some details about him/her-self in the next page, such as email address, the experience range; 1-5 years, 5-10 and more than 10 years, and the job description that includes; Software engineer, programmer, software architect, QA- testing, project manager, team lead, academic student and academic teacher.

In the classification page, a set of comments appear to the participant selected randomly from the overall comments that the first author classified before for one at a time. A criteria is applied on the comments selection to guarantee that no comment is repeated for the same participant and that the comment is not classified by more than two different participants. The web application provided the participants a summary of the definition of each SATD category. By this, each participant is sure about the type of debt that he or she can chosen from the list that embedding the five types of self admitted technical debt. The participant has an option to skip any unsure comment. The aim of the participant classification is to make the Kappa statistical test [18], and to be sure of the reliability of the new dataset that created (Adataset). Cohen’s kappa coefficient [11] used in other studies with the same labeling method [35] [59].The database schema and website pages show in *Appendix A*.

#### 4.1.3.1 Manual annotation result

The first author classified int total 1513 comments and commits out of 5082. Number of comments that classified 1147 out of 3324, and number of commits classified 366 out of 1758. The first author review all comments and commits, and the name of project was hidden, some comments were name of functions or methods with "TODO", or "FixMe" this kind of comments skipped, other types of comments consists of more than one sentences and and give more than one type of SATD also skipped. Additionally, the comment that clearly do not belong to the five types of SATD was skipped. With regard to commits,in addition to what was done in comments, some commits were describe the issue solved by person, it was skipped. Tables 4.1.3 4.1.4 summarize the result of first author comments and commits classification.

Project	No of comment	Classified	Requirement	Design	Defect	Test	Documentation
Camel	1162	452	127	193	52	71	9
Gerrit	172	46	20	15	9	2	0
Log4j	91	23	8	11	4	0	0
Tomcat	1052	334	75	168	58	27	6
Hadoop	625	182	38	77	34	33	0
K9 App	145	75	31	18	18	6	2
WordPress App	77	35	13	14	7	1	0
<b>Total</b>	<b>3324</b>	<b>1147</b>	<b>312</b>	<b>496</b>	<b>182</b>	<b>140</b>	<b>17</b>

FIGURE 4.1.3: Comments classification

Project	No of commits	Classified	Requirement	Design	Defect	Test	Documentation
Camel	739	193	27	34	17	106	9
Gerrit	145	13	4	6	3	0	0
Log4j	74	14	1	9	1	2	1
Tomcat	485	115	12	35	14	48	6
Hadoop	315	31	8	6	5	12	0
<b>Total</b>	<b>1758</b>	<b>366</b>	<b>52</b>	<b>90</b>	<b>40</b>	<b>168</b>	<b>16</b>

FIGURE 4.1.4: Commits classification

### 4.1.3.2 Kappa Test

To mitigate the chance of creating a biased of dataset, a group session for five participants was conducted, one of them has master degree in software engineering and working as team leader with 10 years experience in software development, two participants studies software engineering in Birzeit University and working as a software developer with 6 and 8 years experience. The last two participant has a Bachelor in computer science with 4 and 13 years experience. The group session toke three hour, in the first hour the introduction about SATD and the five types was presented with given more than on example for each type. In the next hour, the discussion was opened to answer questions. Lastly the website viewed and explained the steps for classification. Only software engineering participants interact with the topic, and classified 260 comments and commits that generated randomly from the comments and commits classified before by the first author. The first expert classified 121 comments and 28 commits, the second expert classified 35 comments and 15 commits, the last expert classified 49 comments and 12 commits. Most of the difference between author and experts was between requirements and design, following tables 4.1.5, 4.1.6, 4.1.7, 4.1.8, 4.1.9 shows the result of experts classification comparing with author.

	No of Comments	Requirement	Design	Defect	Test	Documentation
Author	57	57	0	0	0	0
Expert 1	34	26	8	0	0	0
Expert 2	10	6	3	1	0	0
Expert 3	13	11	1	1	0	0
Total Exp.	57	43	12	2	0	0

FIGURE 4.1.5: Experts versus author for requirements classification

To evaluate the level of agreement between both experts and author Cohen's

	No of Comments	Requirement	Design	Defect	Test	Documentation
Author	106	0	106	0	0	0
Expert 1	58	3	53	2	0	0
Expert 2	20	2	16	2	0	0
Expert 3	28	2	23	3	0	0
Total Exp.	<b>106</b>	7	92	7	0	0

FIGURE 4.1.6: Experts versus author for Design classification

	No of Comments	Requirement	Design	Defect	Test	Documentation
Author	49	0	0	49	0	0
Expert 1	28	0	3	24	1	0
Expert 2	11	0	0	11	0	0
Expert 3	10	0	0	10	0	0
Total Exp.	<b>49</b>	0	3	45	1	0

FIGURE 4.1.7: Experts versus author for Defect classification

kappa coefficient [11] calculated. The Cohen's Kappa coefficient is a widely used method to evaluate inter-rater agreement level for categorical scales, and it calculates the proportion of agreement that is chance-corrected. The result of coefficient is scaled from -1 and +1, with a negative value indicating worse than chance agreement, zero means exactly chance agreement, and a positive value indicates better than chance agreement [17]. Whenever the value closer to +1, the agreement is stronger. The level of agreement calculated between two observers (author and experts), and five categories (requirement, design, defect, test, documentation). The online kappa calculator used <sup>2</sup>. The test achieved a level of agreement measured between the author and experts +0.82 based on a sample including of 0.17 of all technical debt types, which is considered almost perfect agreement according to Fleiss [18] values larger than +0.75 are characterized as excellent agreement. Additionally, the test results present the result on PhD expert in statistical field, who recommended the result. Table 4.1.10 present

<sup>2</sup><https://www.graphpad.com/quickcalcs/kappa1/>



	No of Comments	Requirement	Design	Defect	Test	Documentation
Author	38	0	0	0	38	0
Expert 1	25	0	0	0	25	0
Expert 2	7	0	0	1	6	0
Expert 3	6	1	0	0	5	0
Total Exp.	<b>38</b>	1	0	1	36	0

FIGURE 4.1.8: Experts versus author for Test classification

	No of Comments	Requirement	Design	Defect	Test	Documentation
Author	10	0	0	0	0	10
Expert 1	4	0	0	0	0	4
Expert 2	2	0	0	0	0	2
Expert 3	4	0	0	0	0	4
Total Exp.	<b>10</b>	0	0	0	0	10

FIGURE 4.1.9: Experts versus author for Documentation classification

the input data for Kappa test. Each cell in the table is defined by its row and column. The rows specify how each SATD type was classified by the author. The columns specify how the experts classified the subjects. For example, in the second row of the first column 10 comments classified by the author as design, but the experts classified into the requirement. In the second column of the second row 89 comments classified by both observers into the design.

		Experts					
Author		Requirement	Design	Defect	Test	Documentation	Total
	Requirement	43	12	2	0	0	<b>57</b>
	Design	10	89	7	0	0	<b>106</b>
	Defect	0	3	45	1	0	<b>49</b>
	Test	1	0	1	36	0	<b>38</b>
	Documentation	0	0	0	0	10	<b>10</b>
<b>Total</b>		<b>54</b>	<b>104</b>	<b>55</b>	<b>37</b>	<b>10</b>	<b>260</b>

FIGURE 4.1.10: Input data for Kappa test

#### 4.1.4 Data exploratory and analysis

The exploration of data and analysis is one of the important phases for NLP and machine learning. Data visualization helps in deep understanding of the text, and the relations between words, to direct us which tools and techniques should be chosen in the workflow of machine learning and NLP. This section will discuss major techniques that can be used to understand the text data. The new dataset that was created and labeled in this study was combined with the previous dataset that was used in most of the previous study as mentioned above. Most of the processes of data analysis that will be on the raw data without any pre-processing.

##### 4.1.4.1 Data analysis at text level

This technique is considered simple, but it gives insight into the data. The most common methods include; word frequency, sentence length, average length of word, and other representation techniques. After the dataset was extended by 1513 new comments and commits, the total number of sentences is 5585. The first thing will be to look at the number of characters, number of words, and average word length in each sentence. This will provide us with a rough estimate about the length of comments. As figure 4.1.11 shows, the comments generally, between 6 to approximately 700 characters, figure 4.1.12 shows that comments range from 1 to 957 words before any preprocessing, mostly falls between 1 to 150 words. After the comments enter the preprocessing pipeline, the length of sentences range from 1 to 500, mostly falls between 1 and 50 words, and the average of numbers of words in comment is 11, as figure 4.1.13. The average word length ranges between 3 to 10, most of the comments length is 5. Does it mean that the developers write the comments using short words. Figure 4.1.14 presents average word length. To explore the words length it should consider

the stop words that are most commonly used in the language such as "a", "an", "the" etc. Stop words are probably small in length and effect on the data analysis, and may have caused the figure 4.1.14 to be left-skewed.

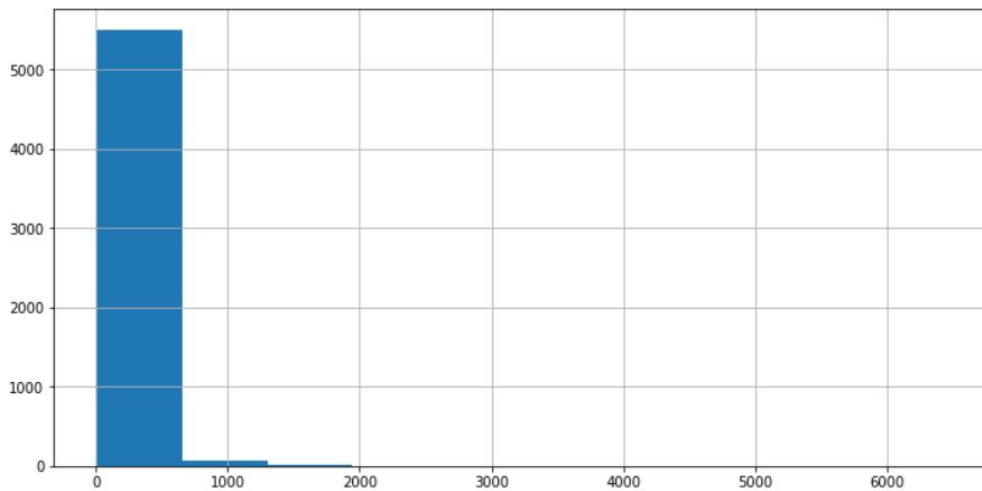


FIGURE 4.1.11: Number of characters appearing in each comment without preprocessing

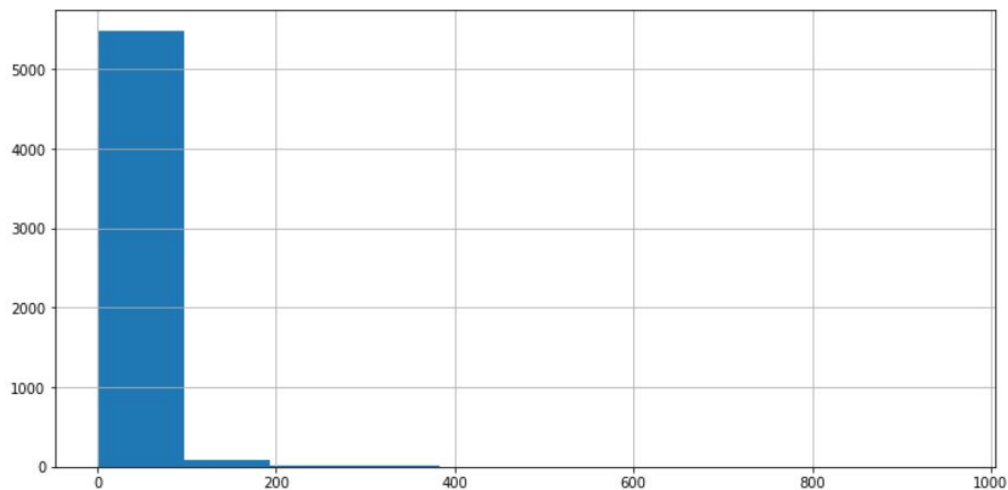


FIGURE 4.1.12: Number of words appearing in each comment without preprocessing

The number of stop words in the data 136 words, the words "the" frequency more than 4800 time. As figure 4.1.15 shows the frequency of them appearing in

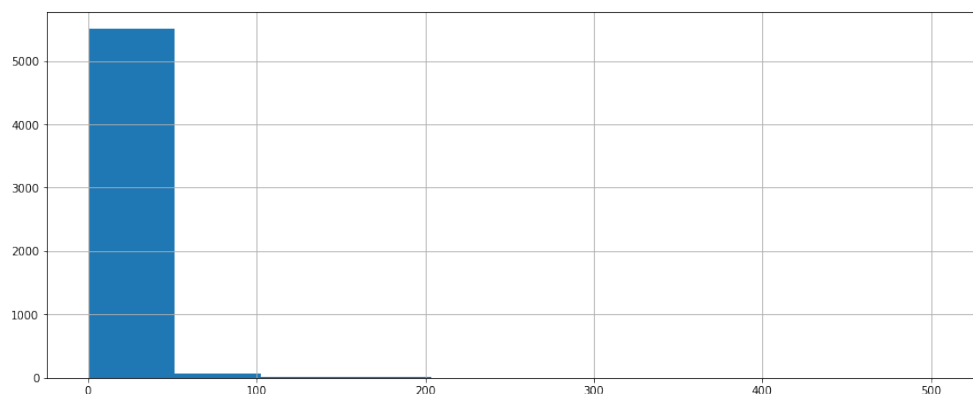


FIGURE 4.1.13: Number of words appearing in each comment after preprocessing

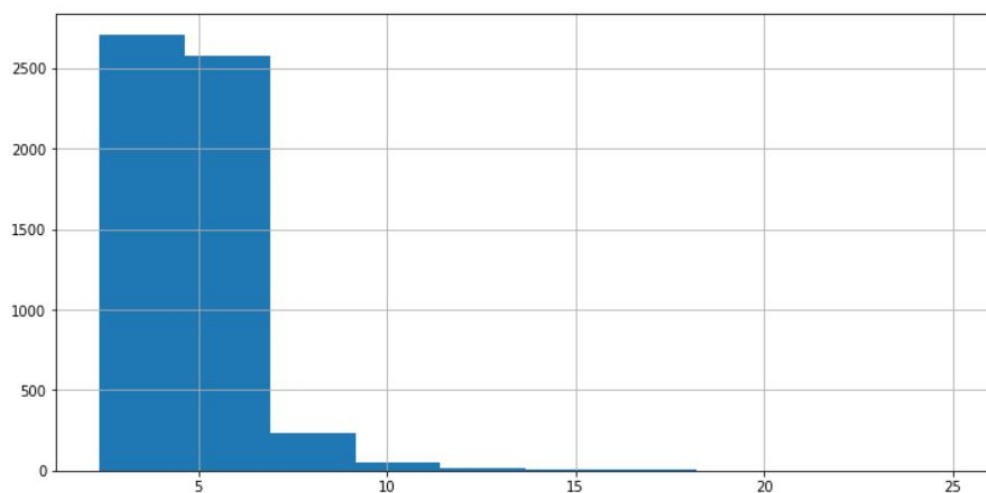


FIGURE 4.1.14: Average word length appearing in each comment

all corpus. After removing stop words, the histogram is different and the graph skewed to the right. Figure 4.1.16 shows the different of the length of words after remove stop words comparing with 4.1.14. Some words reach to more than 15, this because some of words are name of class or method (e.g. "getOut(boolean)").

As a result stop words effect on the sentence in features extraction process. After remove stop words, the inspection of which words appear other than these

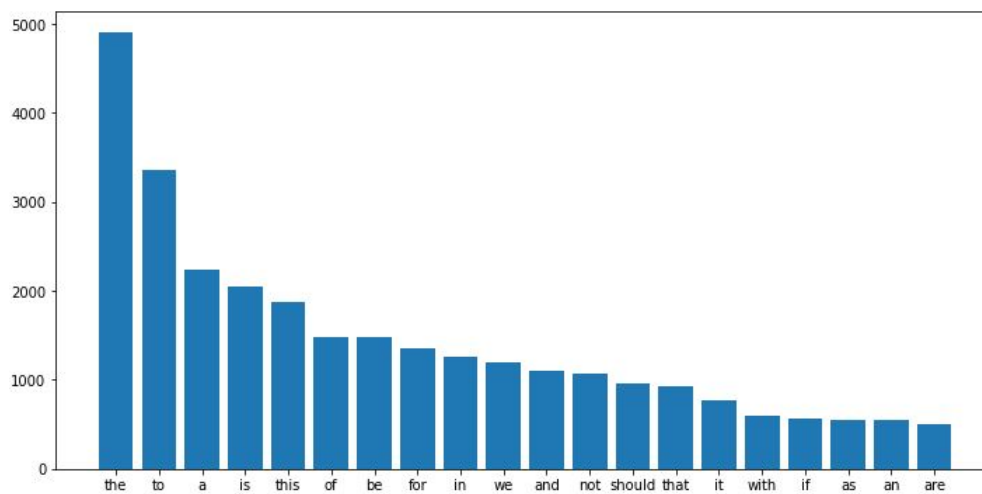


FIGURE 4.1.15: Frequency of stop words

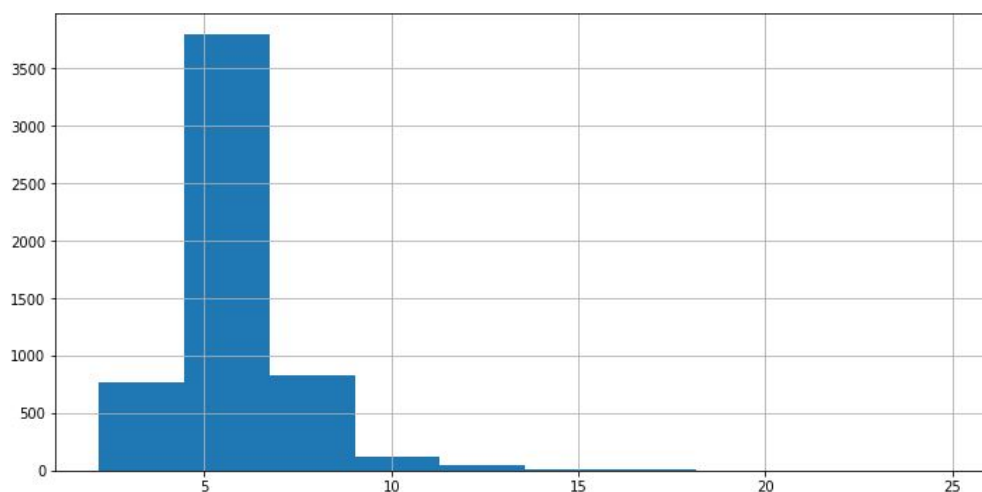


FIGURE 4.1.16: Average word length appearing in each comment after stop words removed

stop words frequently in figure 4.1.17 shows the most of words are punctuation marks, these symbols use frequently because it is the start of comment in source code file to avoided by the compilers. But these symbols did not add any meaning for the sentences. Additionally, the word "TODO" appearing frequently, so it will remove from the comments to view the other words frequently, and it will not remove during feature extraction phase. Figure 4.1.18 shows the frequency

of words after remove punctuation marks.

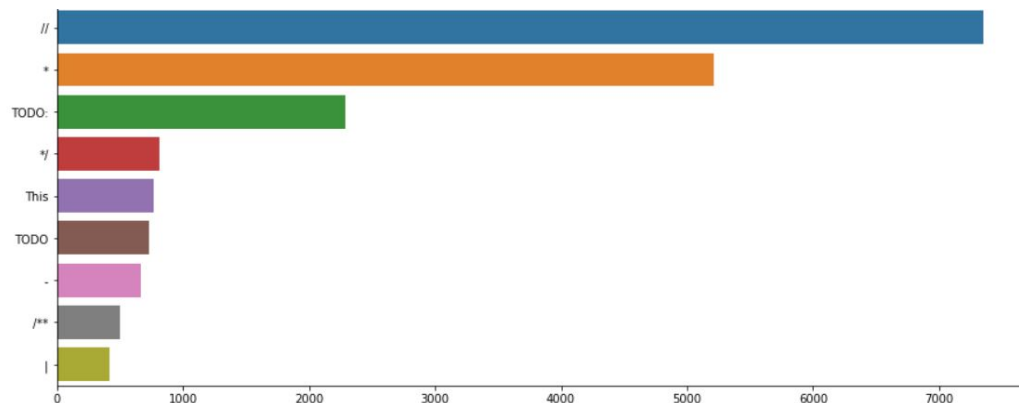


FIGURE 4.1.17: Frequency of words

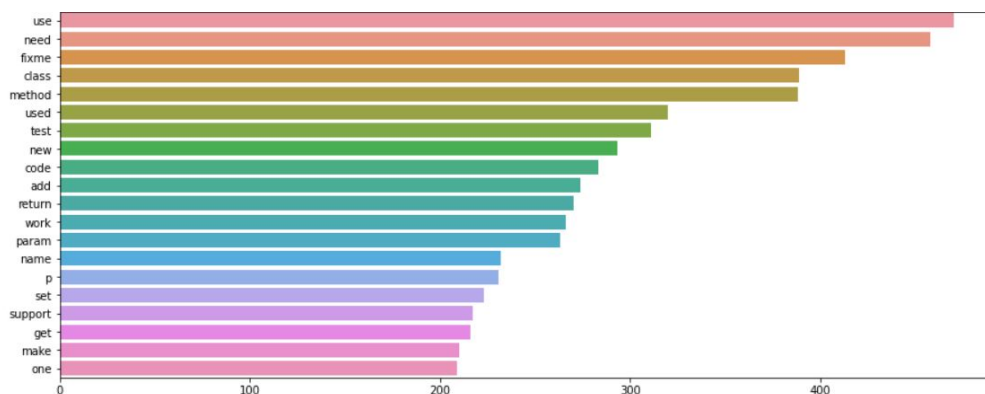


FIGURE 4.1.18: Frequency of words after remove punctuation marks

Word cloud is a great way to represent text data, and give overview about the most words frequency or importance. Figure 4.1.19 shows the most importance words and frequency after clean the comments from punctuation and stop words.

#### 4.1.4.2 Data analysis at features level

After the nature of comments has been understood as a text, the exploration of the data and analysis will be at a features level, when the words converted to



network for top 200 words. The measure of strong co-occurrence between words is the words are connected with lines (edges), words plotted close to each other do not necessarily imply that they have a strong co-occurrence. The thicker line (edge) between two words indicates that these two words have a stronger co-occurrence, according to the value of its Jaccard coefficient. This network reflects the overview that how the same words are share in the different types of SATD. As a result that the count-based approach such as TF-IDF may not the best method for feature extraction, and may need to include the word embedding techniques such as word2vec with deep learning to satisfy a good result.

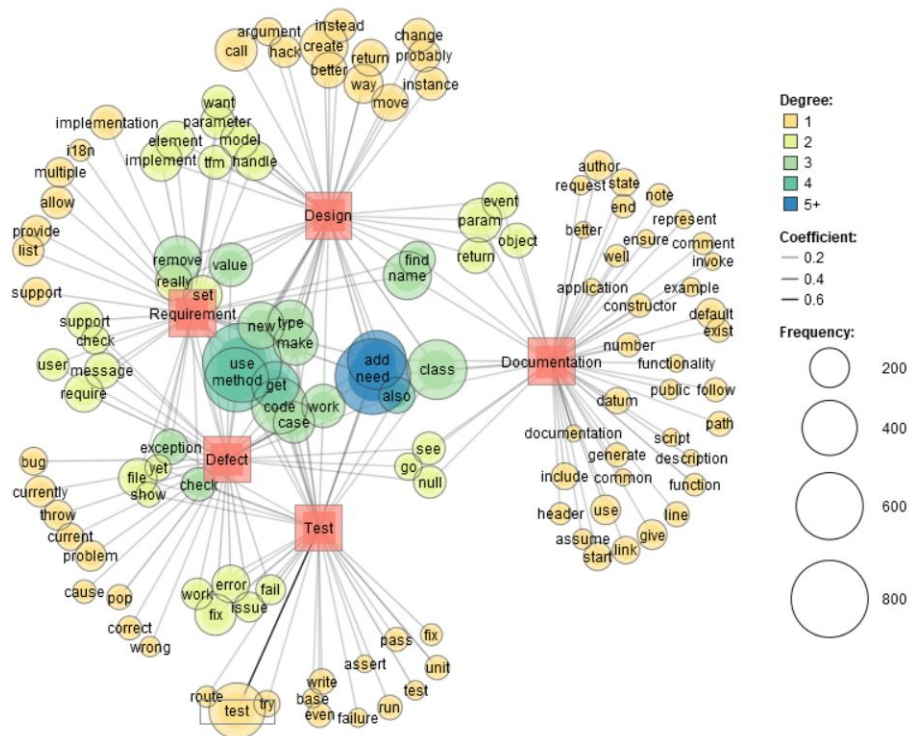


FIGURE 4.1.20: Co-occurrence network of words



## 4.2 Research approach

An empirical study was conducted between dependent and independent variables, to measure the relationship between the models and machine learning techniques for the accuracy of SATD classification. Two independent variables considered; pre-trained models and machine learning algorithms, and one dependent variable; the classification accuracy. To do the experiment and after preparing the dataset, the Colab<sup>4</sup> will be used that provided by the Google and Tensorflow<sup>5</sup>, the experiment will be done through three main phases; preparing the comments by pre-processing the text using NLP techniques, feature engineering to converting tokens of text into features, and classification phase using machine learning approach.

## 4.3 System design.

### 4.3.1 Preprocessing.

The aim of the preprocessing phase is to focus more on the words of a sentence that give the meaning. Usually, the comments and commits are written in natural language, and it includes noises that don't affect on the semantic meaning of sentence. These sentences need to handle through pipeline processes, to keep the important words and remove the noise words, that will be form the features input into machine learning algorithms. Some of NLP techniques<sup>6 7</sup> used to perform this phase as following:

---

<sup>4</sup><https://colab.research.google.com>

<sup>5</sup><https://www.tensorflow.org>

<sup>6</sup><https://www.nltk.org>

<sup>7</sup><https://spacy.io/>

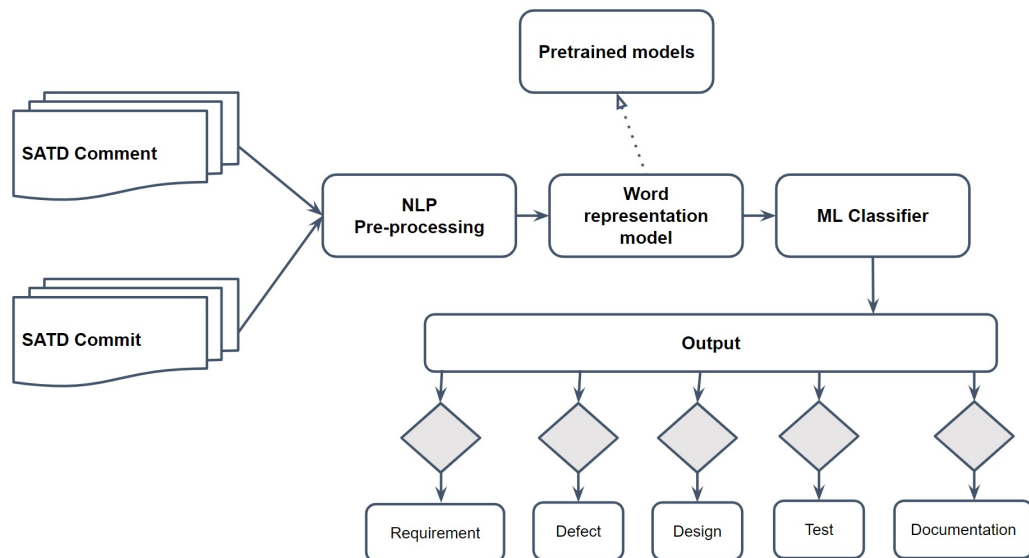


FIGURE 4.3.1: System Design

#### 4.3.1.1 Tokenization.

Tokenization is one of the first steps in NLP pipeline preprocessing, and it's the task of split the text into units called tokens with semantic meaning. In English language, the sentences are composed of words split by white-space, each words have a semantic meaning. The following example of raw comment and tokenized it without any preprocessing

##### **E.g Comment :**

```
"/ / I hate to admit it, but we don't know what happened // here.
Throw the Exception."
```

##### **After the comment tokenized :**

```
['/ /', 'I', 'hate', 'to', 'admit', 'it', ',', 'but', 'we', 'do', 'n't', 'know',
'what', 'happened', '//', 'here', ',', 'Throw', 'the', 'Exception', '.']
```

As noticed from the example above how the tokenization split the comment depending on the meaning not on the space-white between the words. For example "don't" splitted into two words "do" and "not". Additionally, the comment

has noises tokens should be cleaned from the sentence, that what will done in the next process.

#### 4.3.1.2 Text cleaning:

The objective of the cleaning steps to removing all undesirable content from comments and commits sentences, by applying the following techniques.

- **Punctuation removal :** In this step all symbols such as backslash , question marks, comma , etc, were removed from the comments sentences.

```
['I', 'hate', 'to', 'admit', 'it', 'but', 'we', 'do', "n't", 'know', 'what', 'happened', 'here', 'Throw', 'the', 'Exception']
```

- **Stop words removal :** Stop words such as but, we, what, etc.

```
['hate', 'admit', 'know', 'happened', 'throw', 'exception']
```

- **Non-alphabetic removal :** Remove the numbers in the text that not give any meaning.

```
['TODO', 'workaround', 'source', 'check', 'failure', 'afterProperties-Set', 'method', 'Executable', 'statement', 'count', '101', 'max', 'allowed', '100']
```

#### After remove non-alphabetic

```
['TODO', 'workaround', 'source', 'check', 'failure', 'afterProperties-Set', 'method', 'Executable', 'statement', 'count', 'max', 'allowed']
```

#### 4.3.1.3 Normalization:

Normalization is a process to convert the words to a more uniform sequence.

Three NLP techniques used for the transformation :

- **Case folding :** In this step all words(token) converting to small letter

- **Parts of Speech (POS)** : Part of Speech tagging is a process of classifying words into their parts of speech (e.g. noun, verb, adverb, adjective etc.) for each word in a sentence. This step important for lemmatization process that return each word to its root with pre-knowledge the POS tags, in order to maintain the meaning.
- **Lemmatization** : The Spacy lemmatizer comes with pretrained models that can provide various properties of the text, such as POS tags, named entity tags, so the POS tags did not needed, since they explicitly provided like NLTK that uses the WordNet lemmatizer library.

**Comment before preprocessing**

```
// I hate to admit it, but we don't know what happened // here.  
Throw the Exception.
```

**Comment after preprocessing**

```
['hate', 'admit', 'know', 'happen', 'throw', 'exception']
```

### 4.3.2 Features engineering

The second phase in the study approach is to extract useful information from comments, and representation them into a form suitable for machine learning. Two strategies will be used for transformation the words to math form (numbers). The first strategy that include the NLP techniques and depend on **syntax**. This strategy defines the grammatical structures or the set of rules defining a language, such as Bag-of-words and TF-IDF. The second strategy that focus on the **semantics** of words that takes care of the meanings, and how to collect the words together to make sense with consider the syntactical rules, this strategy also called word embedding method, 5 techniques will be used in this study ; Word2vec, software engineering word2Vec model, Fasttext, BERT, and Glove.

Additionally, Universal Sentence Encoder (USE) used to convert all sentence to vectors with height dimension without split it into words.

#### 4.3.2.1 Syntactic vectorization methods

- **Bag-of-Words (BOW)** : is a simple representation method, that use the frequency of the words in the particular document, this method also called Term frequency (TF). For understanding BOW for example, after pre-processing the comments, list of vocabulary from all comments is generated , each comment represented as a numeric vector. The length of each vector is equal to the size of the vocabulary list. Every entry in the vector corresponding a word in the vocabulary list, the value of entry would be equal the frequency of word in particular comment, or zero if the word not exist in that comment. All the vectors reshaped in the matrix with dimensions (number of comments \* size of vocabulary list). The following example shows BOW matrix for only tow comments after cleaning and normalization.

**Comment 1** : ['check', 'load', 'consecutive', 'run', 'end', 'outofmemoryerror', 'fail', 'native', 'library', 'load', 'time', 'far', 'perfect', 'work', 'case']

**Comment 2** : ['todo', 'warning', 'line', 'show', 'code', 'contain', 'variable', 'error', 'cause', 'trouble', 'parser', 'definitely', 'well']

	case	cause	check	code	consecutive	contain	definitely	end	error	fail	far	library	line	load	native	outofmemoryerror	parser	perfect	run	show	time	todo	trouble	variable	warning	well	work
Comment 1	1	0	1	0	1	0	0	1	0	1	1	1	0	2	1	1	0	1	1	0	1	0	0	0	0	0	1
Comment 2	0	1	0	1	0	1	1	0	1	0	0	0	1	0	0	0	1	0	0	1	0	1	1	1	1	1	0

FIGURE 4.3.2: BOW vectors for tow comments

As the figure 4.3.2 shown for each comment the vector with length equal the length of total words of vocabulary list. BOW method extract the features depending on the frequently of word, at the same time it do not

take the order of words in sentence. N-grams is a method for enhanced the BOW that can take range of words that represent entries of the vector. Figure 4.3.3 show the same example aforementioned after applying N-grams method with rang(1,3), which it means for each entry it pick the neighbors until 3 words. This increase the length of vector, and to avoid **overfitting** that occur when the vector length overcome the vocabulary list. **max\_features** parameter will build a vocabulary such that the size of the vocabulary would be less than or equal to **max\_features** ordered by the frequency of tokens in a corpus. Other issue may face the N-grams method called **underfitting**, that occur when the phrases might occur very frequently in an individual comment or may be present in almost all comments in the corpus. For avoiding this issue **max\_df** as threshold used to ignore terms having a document frequency higher than that threshold. Moreover, **min\_df** uses to remove the terms that occur fewer times in a document than a given threshold.

	case	cause	cause trouble	cause trouble parser	check	check load	check load consecutive	code	code contain	code contain variable	consecutive	consecutive run	consecutive run end	contain	....	warning	warning line	warning line show	well	work	work case
<b>0</b>	1	0	0	0	1	1	1	0	0	0	1	1	1	0	....	0	0	0	0	1	1
<b>1</b>	0	1	1	1	0	0	0	1	1	1	0	0	0	1	....	1	1	1	1	0	0

FIGURE 4.3.3: BOW after applying N-gram range between 1 and 3

- **Term frequency inverse document frequency(TF-IDF)** As mentioned in background section. The main difference from the BOW is the TF-IDF take in consideration weight of word. while BOW is only depend on the frequently of words across a document to built the vector. For example the figure 4.3.4 show the representation of tow comments.

The approach of this study adopts the TF-IDF method, because it include BOW with enhancement the value of word representation .

	case	cause	check	code	consecutive	contain	definitely	end	error	fail	far	library	.....	well	work
0	0.24	0	0.24	0	0.24	0	0	0.24	0	0.24	0.24	0.24	.....	0	0.24
1	0	0.28	0	0.28	0	0.28	0.28	0	0.28	0	0	0	.....	0.28	0

FIGURE 4.3.4: TF-IDF representation for two comments

### 4.3.2.2 Word embedding method

The count-based feature engineering strategies for transformation text into vectors was discussed. The models that used belonging to a family of Bag of words as aforementioned. While they are effective models for extracting features from comments, due to the representation model being just a bag of unstructured words, and to avoid loss the important information like the semantics, structure and the context around nearby words in each comments. This motivates us to explore state of the art models to capture this important information that embedding in the representation of words.

- **Word2Vec:** is unsupervised deep learning model provided by the Google, it used to train word embedding, or vector representation of words. In this study two models belong to Word2Vec family will be used, the first one is Word2Vec model<sup>8</sup> trained on part of Google News dataset (about 100 billion words). The model contains 300-dimensional vectors for 3 million words and phrases. The second one, is a software engineering specific model[16]. It is a word2vec model trained over 15GB of textual data from Stack Overflow posts, 6 billion of words used for training task, the output pre-trained . The following example 4.3.5 shows the similarity between six words using these models.
- **GloVe :** is a extended of Word2vec model, GloVe works similarly as Word2Vec. Word2Vec is a "predictive" models that predicts context given a word, and learns

<sup>8</sup><https://code.google.com/archive/p/word2vec/>

Model	Result
Google Word2vec model	<pre>print(word_vec.similarity('function', 'method')) print(word_vec.similarity('class', 'object')) print(word_vec.similarity('object', 'instance')) 0.24744253 0.08910332 0.19355416</pre>
Software Engineering domain model	<pre>print(word_vec2.similarity('function', 'method')) print(word_vec2.similarity('class', 'object')) print(word_vec2.similarity('object', 'instance')) 0.74832416 0.6714444 0.7376088</pre>

FIGURE 4.3.5: Word2vec models for different domain

their vectors to enhance their predictive ability of Loss. GloVe<sup>9</sup> is a count-based model. It learns by building a co-occurrence matrix (words X context) that essentially count the number of times the word appears in context, in order to reduce the dimensionality on the co-occurrence counts matrix. The public domain model used "glove.840B.300d", that includes 840B tokens, 2.2M vocabulary, 300-dimensional vectors, and the model size is 2.03GB.

**Fasttext:** Fasttext is a library introduced by the Facebook for efficient learning of word embedding and text classification. The model used "crawl-300d-2M.vec": 2 million word vectors trained on Common Crawl (600B tokens) with 300 dimensional vector<sup>10</sup>.

**BERT :** stands for Bidirectional Encoder Representations from Transformers (BERT). BERT<sup>11</sup> is introduced in two variants, such as BERT-BASE and BERT-LARGE. The BERT-BASE has a number of transformer blocks 12, hidden layer size 768 ,

<sup>9</sup><https://nlp.stanford.edu/projects/glove/>

<sup>10</sup><https://fasttext.cc/docs/en/english-vectors.html>

<sup>11</sup><https://github.com/google-research/bert>



attention heads 12 and total parameters 110M. The BERT-LARGE has a number of transformer blocks 24, hidden layer size: 1024, attention heads 16 and Total parameters 340M. ktrain library <sup>12</sup> used in order to use the BERT model. For BERT parameters tuning. The recommendation method was followed as mentioned in <sup>13</sup>

### 4.3.3 Machine learning Classifiers

The last phase in this study is classifying the comments into one of the considered five categories ( Requirement, Design, Defect, Test, Documentation). More than one algorithm used that belong to the classic machine learning approach, and neural network approach that mostly used with deep learning. Four classification algorithms will be used in this study; Support Vector Machines classifier (SVM), Naive Bayes classifier (NB), Random Forest Classifier (RF) and CNN for classification task.

#### 4.3.3.1 Support Vector Machines classifier (SVM)

In most of the previous studies, the SVM was used as a binary classifier for SATD identification task [20]. In this study, the SVM used as a multi-classifier with the features that explained in the earlier sections. The Scikit-Learn used, which is a <sup>14</sup> Python library that includes SVM implementation.

#### 4.3.3.2 Naive Bayes classifier (NB)

The NB machine learning algorithm is based on Bayes theorem. NB is one of the most famous and successfully applied supervised machine learning classifier

---

<sup>12</sup><https://libraries.io/pypi/ktrain>

<sup>13</sup><https://arxiv.org/pdf/1810.04805.pdf>

<sup>14</sup><https://scikit-learn.org/stable/modules/svm.html#svm>

in the NLP applications [20][28]. In this study, NB will be used as classifier to identify the technical debt classes.

#### 4.3.3.3 Random Forest (RF)

A random forest is a meta estimator that uses averaging to increase predictive precision and control over-fitting by fitting a range of decision tree classifiers on different sub-samples of the dataset. The RF use with default parameters as provided by Scikit-Learn <sup>15</sup>

#### 4.3.3.4 Convolution Neural Network (CNN)

CNN approach is used by more complex and modern forms of Artificial Neural Networks (ANNs). The CNN model used for multi classification task, where the CNN model takes an input comment and predicts the type of SATD out of the five categories (requirement, design, defect, test, documentation). The architecture of the CNN includes an input layer, a convolutional layer, pooling layer, fully connected layer, and finally output layer. The architecture of CNN is described in detail in background chapter 2. The implementation of the CNN model and hyper parameters will be studied and described in details in the next chapters.

Pandya et al. [50] introduced the study to comparing different Deep Neural Network(DNN) that used with NLP. The main finding of this study was the CNN model was very suitable for text and sentimental classification. While the RNN is well suited for sequence modeling. So this study will be adopted the CNN approach with activation function such as "softmax" as recommended by that study.

---

<sup>15</sup><https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>

The following figure 4.3.6 summarizes the three main processes of system design.

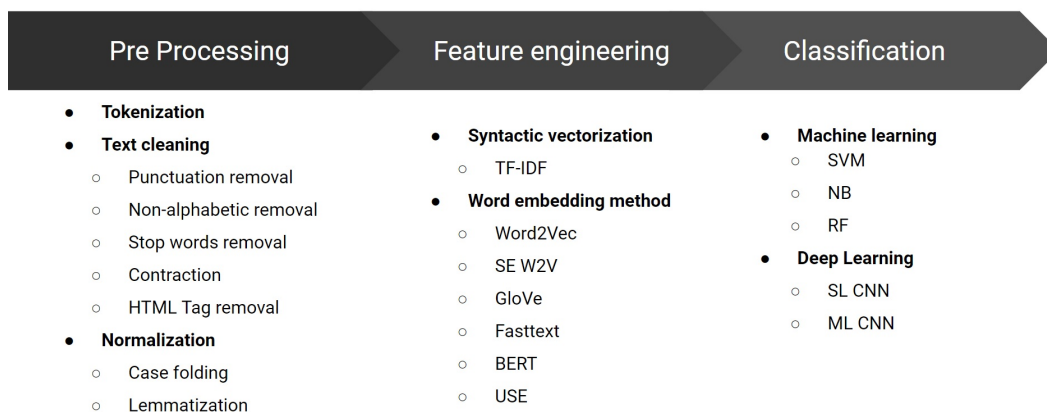


FIGURE 4.3.6: The three main processes in system design

#### 4.4 Evaluation metric:

The proposed system will be evaluated using evaluation metrics, and the definition of the three statistics calculation include:

- **TP (true positive)** : shows the number of comments that predicted to the true classes.
- **FP (false positive)** : represents the number of comments that predicted as false classes and they actually belong to the false classes.
- **FN (false negative)** : the number of comments that actually true classes but the predicted to the false classes

The Precision, Recall, and F1-score computed using these statistics, to evaluate the performance of a classification methods, figure4.4.1 show the confusion matrix for the proposed system. A general rule of thumb (80:20) used for split the dataset, 0.80 for train, and 0.20 of dataset for testing.

**Actual/True**

	Requirement	Design	Defect	Test	Documentation	
Predicted	Requirement	TP	FP <sub>Req</sub> FN <sub>Des</sub>	FP <sub>Req</sub> FN <sub>Def</sub>	FP <sub>Req</sub> FN <sub>T</sub>	FP <sub>Req</sub> FN <sub>Doc</sub>
	Design	FP <sub>Des</sub> FN <sub>Req</sub>	TP	FP <sub>Des</sub> FN <sub>Def</sub>	FP <sub>Des</sub> FN <sub>T</sub>	FP <sub>Des</sub> FN <sub>Doc</sub>
	Defect	FP <sub>Def</sub> FN <sub>Req</sub>	FP <sub>Def</sub> FN <sub>Des</sub>	TP	FP <sub>Def</sub> FN <sub>T</sub>	FP <sub>Def</sub> FN <sub>Doc</sub>
	Test	FP <sub>T</sub> FN <sub>Req</sub>	FP <sub>T</sub> FN <sub>Des</sub>	FP <sub>T</sub> FN <sub>Def</sub>	TP	FP <sub>T</sub> FN <sub>Doc</sub>
	Documentation	FP <sub>Doc</sub> FN <sub>Req</sub>	FP <sub>Doc</sub> FN <sub>Des</sub>	FP <sub>Doc</sub> FN <sub>Def</sub>	FP <sub>Doc</sub> FN <sub>T</sub>	TP

FIGURE 4.4.1: confusion matrix for the five SATD classes

- **Precision** : measure the percentage of comments that are correctly classified as true classes among all comments classified as true classes.

$$Precision = \frac{TP}{TP + \sum FP} \quad (4.1)$$

- **Recall** : denotes the ratio of all comments that are correctly classified into true classes.

$$Recall = \frac{TP}{TP + \sum FN} \quad (4.2)$$

- **F1-score** : is a harmonic mean of precision and recall.

$$F1 - score = \frac{2 * precision * recall}{precision + recall} \quad (4.3)$$

$$Overall\ accuracy = \frac{\sum TP}{\sum TP + \sum FP} \quad (4.4)$$

- **Support** : is a number of test samples for each SATD class.

- **macro average** : macro average compute precision, recall, and F1-score for each label, and returns the average without considering the proportion for each label in the dataset.
- **weighted average** : compute precision, recall, and F1-score for each label, and returns the average considering the proportion for each label in the dataset.

## Chapter 5

### Experimental setup

The main goal of this research is to investigate the effectiveness of NLP techniques, Machine Learning( ML), and Deep Learning(DL) approaches for classification SATD extracted from source code comments and commits.

In order to achieve this goal, a set of experiments will be conducted using collected dataset and available datasets, with a combination of NLP and ML techniques. Each ML classifier is used with three datasets. The first dataset is that introduced by Maldonado (will be referred to as the Mdataset though out the remaining of this thesis). The second dataset is that created in this study (will be referred to as the Adataset). The third dataset is a merge of Adataset and the Mdataset. the experiments are designed and conducted in order to give answers of the research questions which are presented in the introduction. Experiments with various feature engineering techniques have been conducted to present an answer to the (RQ1): 'How well the NLP pre-trained models can improve the identification of self-admitted technical debt from source code comments and commits effectively?'. The following features engineering techniques will be used:

- Term frequency inverse document frequency(TF-IDF).

- Words embedding that include the following pre-trained models:
  - Universal Sentences Encoder (USE).
  - Word2Vec
  - Global Vectors for Word Representation (GLOVE)
  - Fasttext
  - Bidirectional Encoder Representations from Transformers (BERT)

Similarly, a various combination of machine learning techniques (traditional and deep learning) have been used in the experiments in order to present an answer to the second research question (RQ2): 'How well machine learning algorithms that include (SVM, NB, RF, and CNN) can automatically classify the five SATD types efficiently?'. The following machine learning classifiers used:

- Classic machine learning :
  - BernoulliNB Naive Bayes, the Bernoulli NB can focus on a single words, also it can count how many times that words does not occur in the document.
  - Random Forest Classifier
  - Support Vector Machines SVM (LinearSVC)
- Deep learning : To answer the (RQ4): 'Does increasing the numbers of layers in CNN model improve the accuracy of the study approach? '. Two models of CNN were built as the following :
  - Single-layer Convolutional Neural Network (CNN)
  - Multiple-layer Convolutional Neural Network (CNN).

## 5.1 Environment setup:

To conduct the experiments, Google Colaboratory, or “Colab” for short used, which is a cloud service supports GPU processors. Colab allows to write and execute Python in browser. Table 5.1 shows the detailed specifications of the processing capability that used in all of the experiments. Python is used in ML and NLP strategies since it is preferred in this domain over other languages [2]. It includes a large range of modules and libraries for natural language processing (NLP) and data processing [64].

Type	Specification
CPU model	Intel(R) Xeon(R) CPU @ 2.20GHz
Cache size	56320 KB
Ram	13.3 GB
Disk	69 GB
GPU	Tesla T4
OS	Ubuntu 18.04.5 LTS

TABLE 5.1: Environment setup

## 5.2 Pre-Processing

In order to clean and prepare text data in a predictable and analyzable manner for the experiments, a set of pre-processing steps used, as described earlier in the methodology chapter. In all of reported experiments, the following pre-processing steps used.



### 5.2.1 Tokenization

Tokenization is one of the first steps in NLP pipeline preprocessing, and it's the task of split the text into units called tokens. The NLTK <sup>1</sup> tool kit used for tokenizing both the collected dataset (Adataset) and the Mdataset into sentences and words.

### 5.2.2 Text cleaning

In this step all irrelevant data are removed including punctuation, stop-words, non alphabetic terms. The NLTK tool kit and "stripped" used that is python embedded library for this task.

### 5.2.3 Normalization

Normalization is a process to convert the words to a more uniform sequence. Three NLP techniques used for the transformation. First, all tokens converted into small letter. Second, the words classifying into their Parts of Speech (PoS) (e.g. noun, verb, adverb, adjective etc.) for each word in a sentence. Finally, the POS tags to lemmatization process that return each word to it root with pre-knowledge the POS tags. The NLTK used that uses the WordNet lemmatizer library <sup>2</sup>.

## 5.3 Features engineering

Multiple NLP techniques were used to extract features from text. Two approaches were used for converting the words to form that will be understand by machine

---

<sup>1</sup><https://www.nltk.org/>

<sup>2</sup><https://wordnet.princeton.edu/>

learning to classify the text comments. The following techniques are used in all experiments.

### 5.3.0.1 TF-IDF vectorization

After the comments and commits pass through the pre-processing pipeline as mention above. The tokens are converted into numbers (or features). Then these tokens used to create dictionary of unique words for all comments in each dataset. The size of the dictionary for Mdataset was 6327 unique words out of 44895 words. For Adataset, the dictionary size was 3948 uniques words out of 20929 words. The combined dataset has 8390 unique words out of 65824 words. This means, for each comments the vector size is equal to the length of dictionary. These steps repeated to show how much the special terms in the comments such as "TODO", "FIXME", "XXX", and "HACK", take place from all the words. This is shown in the tables 5.2, 5.3. As it is shown from the figures in the tables, the special terms appeared 4112 times in the combined dataset, 1019 times in Adataset, and 3093 times in the Mdataset.

	M DS	A DS	Combined DS
All Words	44895	20929	65824
Unique words	6327	3948	8390

TABLE 5.2: Unique words with special terms

	M DS	A DS	Combined DS
All Words	41802	19910	61712
Unique words	6324	3944	8387

TABLE 5.3: Unique words without special terms

Finally, the TF-IDF method calculates the relative frequency of each unique word in a specific comment. The words are then given a weight that is inversely proportional with their frequency across the whole dataset that is frequently repeated with little weights.

### 5.3.0.2 Word2Vec vectorization

Word2Vec<sup>3</sup> is one of the most popular word embedding pre-trained model introduced by google. Five pre-trained models used in the experiments.

- Word2Vec model pre-trained on part of Google News dataset (about 100 billion words). This model generates 300-dimensional vectors for 3 million words and phrases.
- Software engineering specific model[16]. It is a word2vec model trained on over 15GB of textual data from stack overflow posts. Six billions of words were used for training. The output is 200-dimensional vectors for 1,787,145 keywords.

### 5.3.0.3 Universal sentence encoder

Universal sentence encoder is a family of pre-trained sentence encoders introduced by Google. This method will be used for embedding sentences for classic machine learning, instead of using an average of word embedding such that calculated for each sentence. The universal sentence encoder model is trained on huge data and supports more than 16 languages. The output of this model is 512-dimensional vectors for each sentence. <sup>4</sup>

---

<sup>3</sup><https://code.google.com/archive/p/word2vec/>

<sup>4</sup><https://tfhub.dev/google/universal-sentence-encoder-multilingual/3>

#### 5.3.0.4 GloVe vectorization

GloVe is an extension for the Word2vec model. It learns by building a co-occurrence matrix (words X context) that essentially counts the number of times the word appears in the context. The public domain model "glove.840B.300d" used, that includes 840B tokens, 2.2M vocabulary, 300-dimensional vectors, and the model size is 2.03GB <sup>5</sup>.

#### 5.3.0.5 Fasttext vectorization

For efficient learning of word embedding at a character-level, the Fasttext pre-trained model that used; "crawl-300d-2M.vec": 2 million word vectors trained on Common Crawl (600B tokens) with 300-dimensional vectors <sup>6</sup>.

#### 5.3.0.6 BERT vectorization

The BERT-BASE model is used. It has a number of transformer blocks 12, hidden layer size 768, attention heads 12. Tensorflow hub used to load the BERT pre-trained model <sup>7</sup>.

### 5.3.1 Parameters setting for classifiers

In all the experiments, the dataset is splitted into 0.80 for training and 0.20 for testing.

#### 5.3.1.1 Classic machine learning classifiers

For all experiments that used the classical ML algorithms, the default setting parameters used, as the Scikit-learn library provided <sup>8</sup>.

---

<sup>5</sup><https://nlp.stanford.edu/projects/glove/>

<sup>6</sup><https://fasttext.cc/docs/en/english-vectors.html>

<sup>7</sup>[https://tfhub.dev/tensorflow/bert\\_en\\_uncased\\_L-12\\_H-768\\_A-12/4](https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/4)

<sup>8</sup><https://sklearn.org/>

### 5.3.1.2 Deep learning classifiers

For all deep learning-based experiments, the Keras and TensorFlow library used is an open-source neural-network library written in Python <sup>9</sup>. The embedding dimension parameter was fixed according to the pre-trained model used. Most pre-trained models produce 300 dimensions in all experiments, except the software engineering and the universal sentence encoder pre-trained models that are 200 and 512 dimensions, respectively. The number of target classes was set to 5, that equal to the number of SATD types considered in this study. Two architectural neural networks are used for CNN; simple CNN (single hidden layer), and complex CNN (multiple hidden layers). The difference between the two networks is the number of layers. In single-layer CNN, one layer is used for convolutional and pooling layers, whereas three layers are used for complex CNN. The reset of parameters were fixed as follow: **number of filters: 128, number of classes: 5 , stride: 1, filter size for layers in order: [2,3,4], dropout: 0.2, batch size: 32, number of max epochs: 20.**

**To avoid over-fitting problems in training and to get the optimal number of epochs. The model stops training when a monitored metric has stopped improving. The callbacks <sup>10</sup> early stopping parameter used in the fit model, the callback will stop the training when there is no improvement in the validation loss for five consecutive epochs. The activation functions: "relu" and "softmax" ,optimizer: "adam" . Since the training does not depend on the own embedding, the pre-trained model was used, the trainable parameter set to false, and the own embedding matrix passed in weights parameter.**

---

<sup>9</sup><https://www.tensorflow.org/>

<sup>10</sup>[https://keras.io/api/callbacks/early\\_stopping/](https://keras.io/api/callbacks/early_stopping/)

## Chapter 6

### Experiments and results

This chapter presents all the experiments conducted and the results with a discussion. In all experiments presented in this chapter, the accuracy and weighted average of three metrics, precision, recall and F1-score metrics are used for the system performance.

#### 6.1 Experiments using classical machine learning algorithms

In these experiments presented in this section, each of the classical ML techniques described in the methodology chapter, are trained on the training data and evaluated on the test data of each of the three datasets that include Mdataset, Adataset, and the combined dataset.

##### 6.1.1 Experiment set 1: Adataset

This set of experiments includes the three classical classifiers (RF, SVM, and NB) trained and evaluated on TF-IDF and USE features extracted from Adataset. The results of these six experiments are shown in table 6.1.

Figure 6.1.1 shows the performance of the RF classifier with the TF-IDF (best accuracy system) for each type of SATD categories.

	precision	recall	f1-score	support
Defect	0.808	0.512	0.627	41
Design	0.735	0.942	0.826	103
Documentation	1.000	0.200	0.333	10
Requirement	0.904	0.806	0.852	93
Test	0.900	0.964	0.931	56
accuracy			0.822	303
macro avg	0.869	0.685	0.714	303
weighted avg	0.836	0.822	0.810	303

FIGURE 6.1.1: RF and TF-IDF performance metrics for each type of SATD in Adataset

Adataset					
Naive Bayes					
	Precision	Recall	F1-score	Accuracy	Testing time
TF-IDF	0.703	0.683	0.654	<b>0.683</b>	0.006
USE	0.746	0.723	0.726	<b>0.723</b>	0.011
Random Forest					
TF-IDF	0.836	0.822	0.81	<b>0.822</b>	0.029
USE	0.801	0.779	0.771	<b>0.779</b>	0.018
Support Vector Machines					
TF-IDF	0.826	0.812	0.816	<b>0.812</b>	0.006
USE	0.803	0.792	0.795	<b>0.792</b>	0.004

TABLE 6.1: ML performance metrics with all classifiers for Adataset

As shown in table 6.1, in two experiments (RF and SVM) of the first set that

are performed on Adataset, the TF-IDF vectorization features outperform the USE features. The SVM works better than RF in terms of f1-score and testing computation time. However, RF outperforms the SVM in precision and recall. Moreover, the accuracies of the 10 runs of the SVM classifier ranges from 0.792 to 0.812, where, RF ranges from 0.802 to 0.822. In terms of f1-score, The RF and SVM systems with the TF-IDF outperformed the baseline binary classifier system described in [28], by 0.105 and 0.113, respectively. When comparing with only requirement and design classes (baseline classes), The RF and TF-IDF system outperforms the baseline by 0.144 for average f1-score.

Figure 6.1.2 shows the three classic machine learning classifiers with TF-IDF and USE.

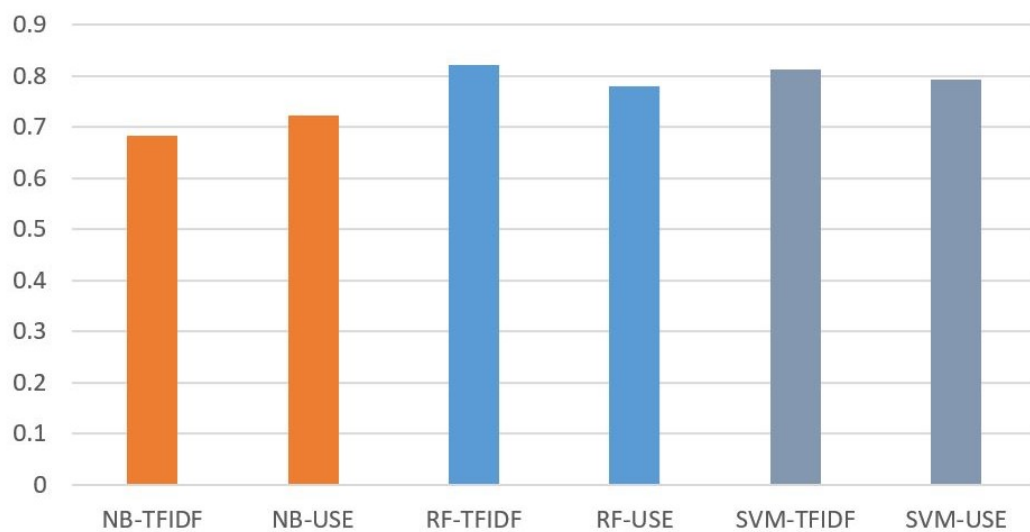


FIGURE 6.1.2: The accuracy of classic machine learning with TF-IDF and USE for Adataset



### 6.1.2 Experiments set 2: Mdataset

This experiments set evaluate the three ML classifiers with Mdataset. All the configurations used in this experiments set is kept the same as used in experiments set 1, except the dataset has been changed to the Mdataset. The main results of the six experiments of set 2 are shown in table 6.2. As it is clear from these results, the RF classifier outperformed the other two classifiers. The figure 6.1.3 shows the results of the RF classifier with TF-IDF vectorization method for each type of the SATD categories.

	precision	recall	f1-score	support
Defect	0.846	0.379	0.524	87
Design	0.813	0.968	0.884	561
Documentation	1.000	0.667	0.800	6
Requirement	0.871	0.536	0.664	151
Test	0.636	0.700	0.667	10
accuracy			0.820	815
macro avg	0.833	0.650	0.708	815
weighted avg	0.826	0.820	0.801	815

FIGURE 6.1.3: RF and TF-IDF performance metrics for each type of SATD in Mdataset

<b>M Dataset</b>					
<b>Naive Bayes</b>					
	<b>Precision</b>	<b>Recall</b>	<b>F1-score</b>	<b>Accuracy</b>	<b>Testing time</b>
<b>TF-IDF</b>	0.67	0.696	0.59	<b>0.696</b>	0.012
<b>USE</b>	0.664	0.639	0.646	<b>0.639</b>	0.018
<b>Random Forest</b>					
<b>TF-IDF</b>	0.826	0.82	0.801	<b>0.82</b>	0.075
<b>USE</b>	0.838	0.807	0.778	<b>0.807</b>	0.035
<b>Support Vector Machines</b>					
<b>TF-IDF</b>	0.774	0.783	0.775	<b>0.783</b>	0.012
<b>USE</b>	0.737	0.753	0.728	<b>0.753</b>	0.007

TABLE 6.2: ML performance metrics with all classifiers for Mdataset

Similar to the experiments set 1, the TF-IDF outperforms the USE vectorization method. By comparing these results with the systems results with Adataset, the Adataset outperforms the Mdataset in the defect, the requirement, and the test categories, whereas, the Mdataset gives better results for the design and the documentation categories. Moreover, The RF system with the TF-IDF outperforms the baseline described in [28] which uses the same Mdataset by 0.055 of average f1-score for two types of SATD (requirement and design). For the five types of SATD, the proposed approach achieves an average of f1-score 0.801, which outperforms the baseline by 0.092 .

Figure 6.1.4 shows the three classic machine learning classifiers with TF-IDF and USE.

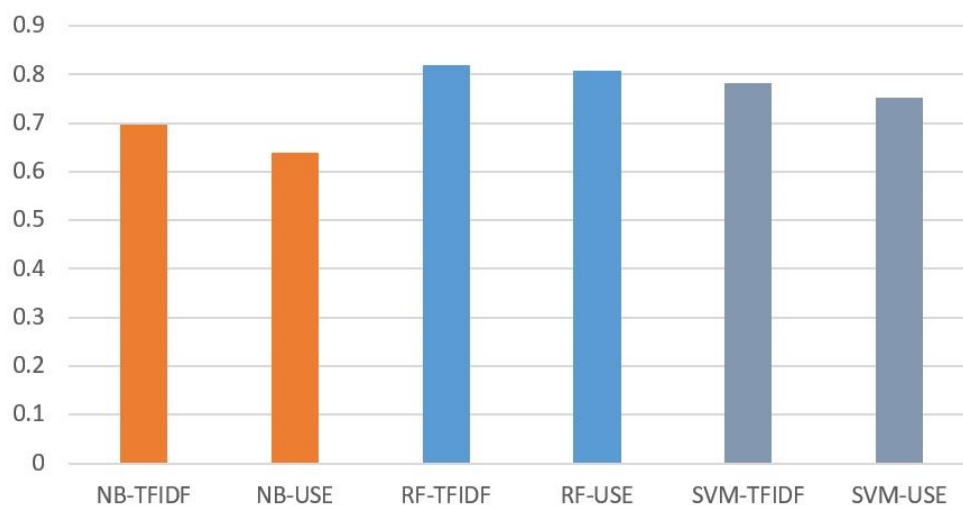


FIGURE 6.1.4: The accuracy of classic machine learning with TF-IDF and USE for Mdataset

### 6.1.3 Experiments set 3: The combined dataset

In this experiments set, Adataset merged with the Mdataset into one combined dataset that used for training and testing the systems presented in this subsection. The main results of the six experiments of set 3 are shown in table 6.3. As it is clear from these results, the RF classifier outperformed the other two classifiers. The figure 6.1.5 shows the results of the RF classifier with TF-IDF vectorization method for each type of the SATD categories.

	precision	recall	f1-score	support
Defect	0.915	0.432	0.587	125
Design	0.781	0.956	0.860	677
Documentation	0.923	0.667	0.774	18
Requirement	0.795	0.484	0.602	217
Test	0.812	0.863	0.836	80
accuracy			0.794	1117
macro avg	0.845	0.680	0.732	1117
weighted avg	0.804	0.794	0.776	1117

FIGURE 6.1.5: RF and TF-IDF performance metrics for each type of SATD in combined dataset

Combined Dataset					
Naive Bayes					
	Precision	Recall	F1-score	Accuracy	Testing time
TF-IDF	0.61	0.627	0.521	<b>0.627</b>	0.016
USE	0.657	0.623	0.633	<b>0.623</b>	0.045
Random Forest					
TF-IDF	0.804	0.794	0.776	<b>0.794</b>	0.093
USE	0.778	0.739	0.699	<b>0.74</b>	0.054
Support Vector Machines					
TF-IDF	0.759	0.767	0.76	<b>0.767</b>	0.015
USE	0.703	0.718	0.698	<b>0.718</b>	0.008

TABLE 6.3: ML performance metrics with all classifiers for combined dataset

The RF outperforms the other classifiers by using the two vectorization methods TF-IDF and USE. By referring back to the two previous experiments set (1 and 2) which used the same techniques with Adataset and Mdataset, the expectation is that the systems performance will be improved by combining the two

datasets, since this gives the systems more training data. However, the results show the opposite, and the accuracy drops from 0.822 and 0.820 for Adataset and Mdataset, respectively, to 0.794 with the combined dataset. In order to get a robust conclusion about the effect of the data combining on the performance, and present an answer to the RQ3 the deep learning techniques with the combined dataset will be used as described in the subsequent sections.

Figure 6.1.6 shows the the three classic machine learning classifiers with TF-IDF and USE.

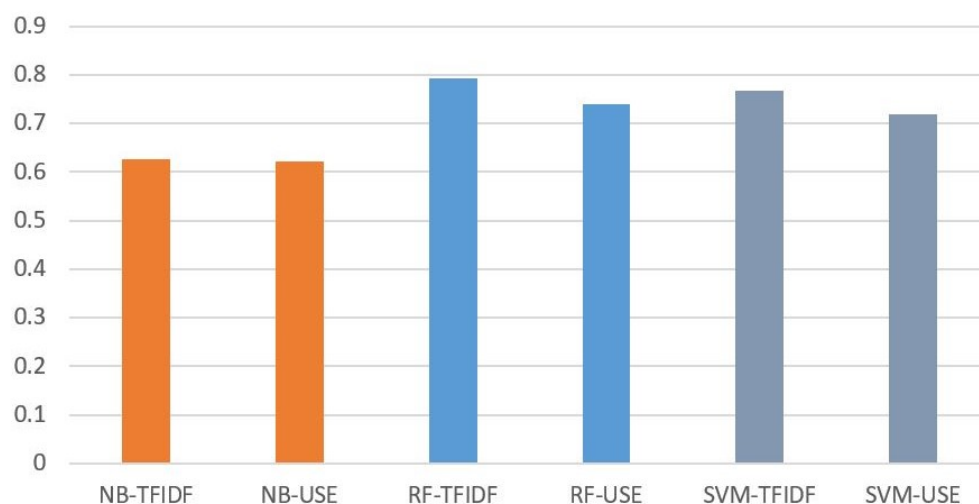


FIGURE 6.1.6: The accuracy of classic machine learning with TF-IDF and USE for combined dataset

## 6.2 Deep learning

In these experiments presented in this section, tow models of CNN are used, single-layer CNN (SLCNN), and multiple-layer CNN (MLCNN). The two models of CNN take the same parameters as mention in the experimental setup section 5. Each model of the CNN classifiers is trained on the train data and evaluated on the test data for each of the three datasets. The five pre-trained models

(W2V, SEW2V, Glove, Fasttext, and BERT), and TF-IDF are all used for features extraction for the CNN models.

## 6.2.1 Single-layer CNN

### 6.2.1.1 Experiments set 4: Adataset

In this set of experiments, the five pre-trained models and TF-IDF used with Adataset to train and test a single-layer CNN. The main results of the six experiments are shown in table 6.4. The results show that the system with the BERT pre-trained model outperformed the TF-IDF and the other pre-trained models. Figure 6.2.1 shows the results of the SLCNN classifier with BERT model for each class of SATD categories.

	precision	recall	f1-score	support
REQ	0.840	0.759	0.797	83
DES	0.832	0.825	0.829	126
DEF	0.568	0.778	0.656	27
TEST	0.967	0.983	0.975	60
DOC	1.000	0.714	0.833	7
accuracy			0.832	303
macro avg	0.841	0.812	0.818	303
weighted avg	0.841	0.832	0.834	303

FIGURE 6.2.1: SLCNN and BERT performance metrics for each type of SATD in Adataset

ADataset						
Single-layer CNN						
	Precision	Recall	F1-score	Accuracy	Testing time	No Epoch
TF-IDF	0.778	0.772	0.771	<b>0.772</b>	0.066	6
W2V	0.817	0.815	0.811	<b>0.815</b>	0.067	6
SE-W2V	0.793	0.785	0.786	<b>0.785</b>	0.066	11
GloVe	0.826	0.822	0.819	<b>0.822</b>	0.066	6
Fasttext	0.835	0.828	0.827	<b>0.828</b>	0.062	6
<b>BERT</b>	0.841	0.832	0.834	<b>0.832</b>	0.107	10

TABLE 6.4: SLCNN performance metrics with all pre-trained models and TF-IDF for Adataset

The results of classical ML classifiers voted to the RF with TF-IDF as a best approach as mentioned in the previous experiments. The RF with TF-IDF will be consider as a baseline for the deep learning experiments. AS the results shows in the above table 6.4. The BERT model is the best classifier that achieves the best accuracy (0.832), and it outperforms the other models in all performance metrics (precision : 0.836, recall : 0.822, f1-score: 0.81, and accuracy: 0.822). The TF-IDF works with less accuracy with deep learning approach. compared with the other pre-trained models, the Fasttext is better than RF, and the GloVe has the same accuracy, but it is better in f1-score than RF. Another observation about the results of experiment set 4 is that CNN with pre-trained models outperform the traditional machine learning.

Figure 6.2.2 shows the single-layer CNN classifier with TF-IDF and pre-trained models.

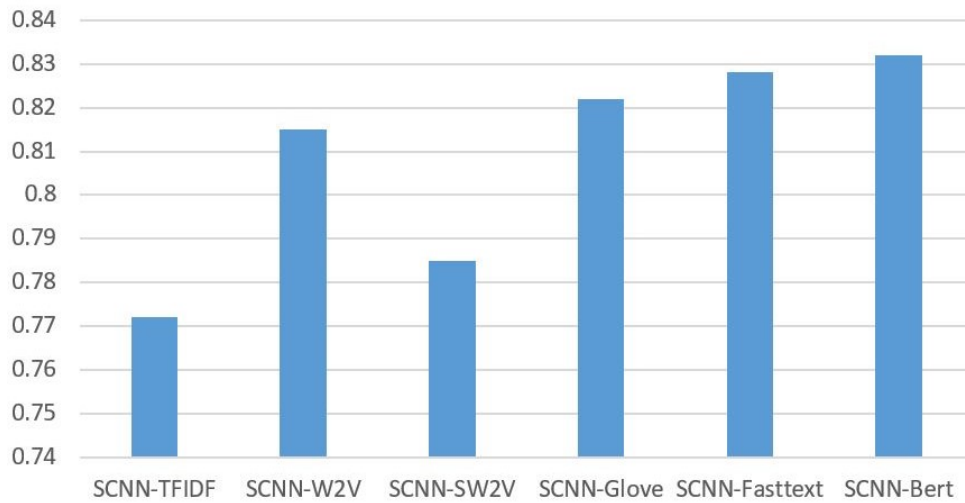


FIGURE 6.2.2: The accuracy of single-layer CNN with TF-IDF and pre-trained models for Adataset

#### 6.2.1.2 Experiments set 5: Mdataset

In this set of experiments, the TF-IDF and five types of pre-trained models are used for features representation with the Mdataset. The main results of the single-layer CNN experiments are shown in table 6.5. These results show, the Word2Vec model outperforms the TF-IDF and the other pre-trained models. Figure 6.2.3 shows the results of the SLCNN classifier with the Word2Vec features, for each type of SATD categories.



	precision	recall	f1-score	support
REQ	0.790	0.550	0.648	151
DES	0.821	0.947	0.879	561
DEF	0.725	0.425	0.536	87
TEST	0.889	0.800	0.842	10
DOC	1.000	0.500	0.667	6
accuracy			0.812	815
macro avg	0.845	0.644	0.715	815
weighted avg	0.807	0.812	0.798	815

FIGURE 6.2.3: SLCNN and Word2Vec performance metrics for each type of SATD in Mdataset

M Dataset						
Single-layer CNN						
	Precision	Recall	F1-score	Accuracy	Testing time	No Epoch
<b>TF-IDF</b>	0.757	0.761	0.739	<b>0.761</b>	0.138	6
<b>W2V</b>	0.807	0.812	0.798	<b>0.812</b>	0.106	7
<b>SE-W2V</b>	0.783	0.791	0.783	<b>0.791</b>	0.118	6
<b>GloVe</b>	0.802	0.802	0.785	<b>0.802</b>	0.156	9
<b>Fasttext</b>	0.804	0.806	0.788	<b>0.806</b>	0.161	9
<b>BERT</b>	0.796	0.804	0.792	<b>0.804</b>	0.224	10

TABLE 6.5: SLCNN performance metrics with all pre-trained models and TF-IDF for Mdataset

In this set of experiments the accuracy of the five pre-trained models ranges from 0.791 to 0.812. The Word2Vec model achieves the best accuracy. By comparing proposed system which classifies SATD into five classes with the result of the binary classification study [55] that used CNN model and the Mdataset, the SLCNN system outperforms it in the five pre-trained models and TF-IDF.

Figure 6.2.2 shows the single-layer CNN classifier with TF-IDF and pre-trained models.

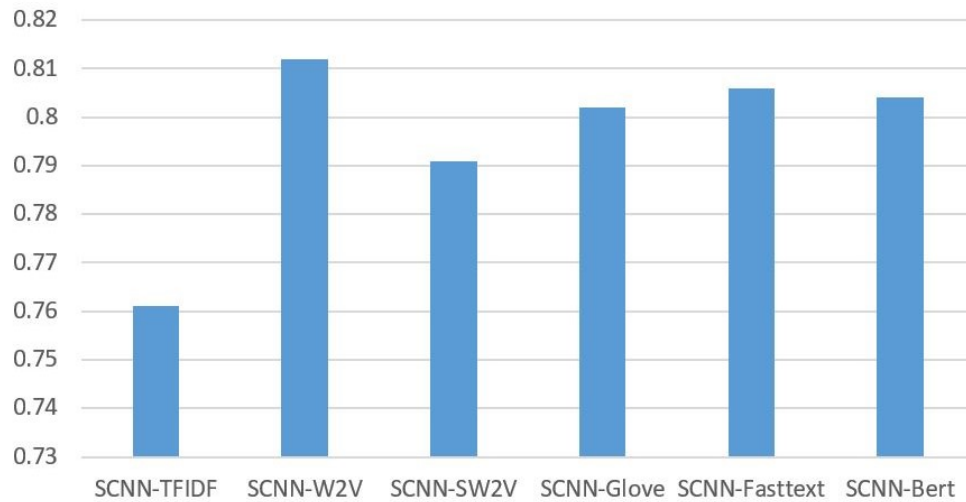


FIGURE 6.2.4: The accuracy of single-layer CNN with TF-IDF and pre-trained models for Mdataset

### 6.2.1.3 Experiments set 6: Combined dataset

The main results of the six experiments of set 6 are shown in table 6.6. The Word2Vec model outperforms the TF-IDF and the other pre-trained models. Figure 6.2.5 shows the results of the SLCNN classifier with Word2Vec model for each type of SATD categories.

	precision	recall	f1-score	support
REQ	0.714	0.530	0.608	217
DES	0.801	0.901	0.848	677
DEF	0.667	0.496	0.569	125
TEST	0.798	0.887	0.840	80
DOC	0.833	0.556	0.667	18
accuracy			0.777	1117
macro avg	0.763	0.674	0.706	1117
weighted avg	0.769	0.777	0.767	1117

FIGURE 6.2.5: SLCNN and Word2Vec performance metrics for each type of SATD in combined dataset

Combined Dataset						
Single-layer CNN						
	Precision	Recall	F1-score	Accuracy	Testing time	No Epoch
TF-IDF	0.726	0.73	0.722	0.73	0.231	6
W2V	0.769	0.777	0.767	0.777	0.224	6
SE-W2V	0.762	0.769	0.76	0.769	0.126	6
GloVe	0.757	0.765	0.752	0.765	0.178	10
Fasttext	0.767	0.774	0.758	0.774	0.13	6
BERT	0.77	0.776	0.77	0.776	0.196	10

TABLE 6.6: SLCNN performance metrics with all pre-trained models and TF-IDF for combined dataset

By these experiments, that are trying to answer the (RQ3): "How well combined the two datasets can improves classification accuracy?". As the results show in the table 6.6, word2Vec is the best. comparing with BERT the result was very close for both model accuracy, and the different between accuracy of two classifiers is 0.001. However, BERT outperforms the word2Vec in f1-score and

precision. More analysis and experiments to explore the effect of data combination on the systems performance are described in section 6.2.2.3 with answer the RQ3.

Figure 6.2.6 shows the single-layer CNN classifier with TF-IDF and pre-trained models.

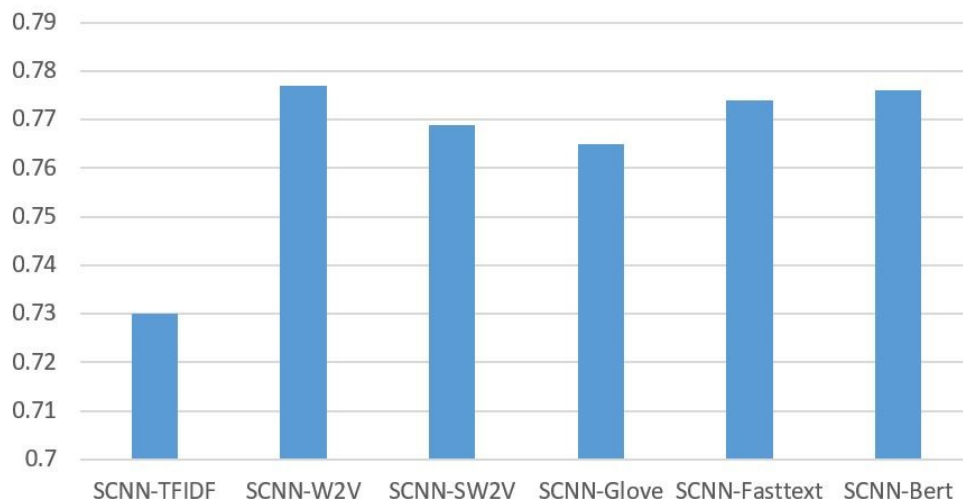


FIGURE 6.2.6: The accuracy of single-layer CNN with TF-IDF and pre-trained models for combined dataset.

### 6.2.2 Multiple-layer CNN (MLCNN)

In order to present an answer to RQ4: "Does increasing the numbers of layers in CNN model improve the accuracy of the study approach ?", a set of experiments are conducted to evaluate MLCNN. All the configurations used in this experiments set is kept the same as used in experiments set SLCNN, except the number of layers has been changed to three.

### 6.2.2.1 Experiments set 7: Adataset

The main results of these six experiments set are shown in table 6.7. These results show, the BERT model outperforms the TF-IDF and the other pre-trained models. Figure 6.2.7 shows the results of the MLCNN classifier with BERT model for each category of SATD.

	precision	recall	f1-score	support
REQ	0.805	0.843	0.824	83
DES	0.857	0.810	0.833	126
DEF	0.613	0.704	0.655	27
TEST	0.952	0.983	0.967	60
DOC	1.000	0.571	0.727	7
accuracy			0.838	303
macro avg	0.845	0.782	0.801	303
weighted avg	0.843	0.838	0.839	303

FIGURE 6.2.7: MLCNN and BERT performance metrics for each type of SATD in Adataset

Adataset						
Multiple-layer CNN						
	Precision	Recall	F1-score	Accuracy	Testing time	No Epoch
TF-IDF	0.789	0.789	0.792	0.79	0.085	8
W2V	0.809	0.815	0.807	0.815	0.148	8
SE-W2V	0.813	0.822	0.814	0.822	0.14	8
GloVe	0.824	0.815	0.816	0.815	0.15	7
Fasttext	0.825	0.828	0.825	0.828	0.142	6
BERT	0.843	0.838	0.839	0.838	0.153	10

TABLE 6.7: MLCNN performance metrics with all pre-trained models and TF-IDF for Adataset

The main results of these experiment set are shown in table 6.7. By comparing the results of the SLCNN and MLCNN systems, the MLCNN outperforms the SLCNN in two models SE-W2V and BERT. Table 6.2.9 and the figure 6.2.10 present the results for SLCNN and MLCNN. For the W2V and Fasttext models, the accuracy and recall are equal, but the f1-score and precision in SLCNN is better than MLCNN. The Golve with SLCNN outperforms the MLCNN.

Figure 6.2.8 shows the multiple-layer CNN classifier with TF-IDF and pre-trained models.

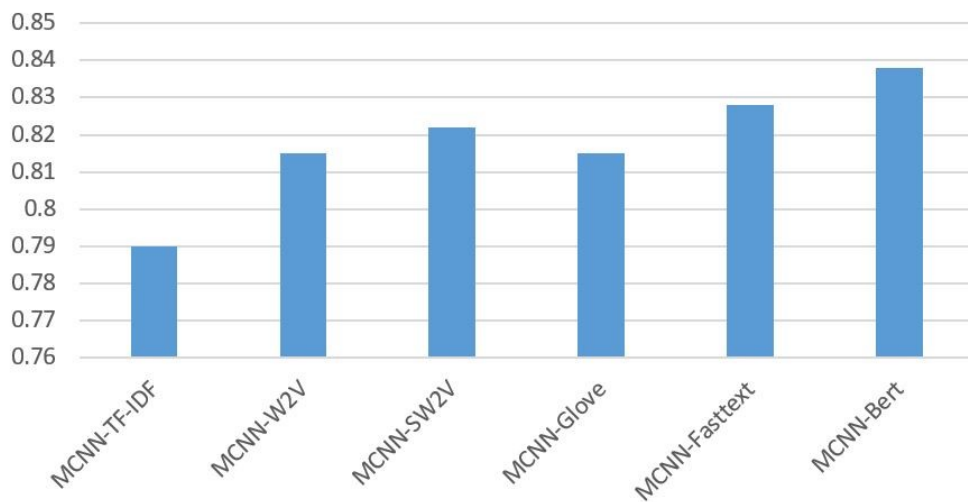


FIGURE 6.2.8: The accuracy of multiple-layer CNN with TF-IDF and pre-trained models for Adataset.

	Single-layer CNN				Multiple-layer CNN			
	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy
<b>W2V</b>	0.817	0.815	0.811	<b>0.815</b>	0.809	0.815	0.807	<b>0.815</b>
<b>SE-W2V</b>	0.793	0.785	0.786	<b>0.785</b>	0.813	0.822	0.814	<b>0.822</b>
<b>GloVe</b>	0.826	0.822	0.819	<b>0.822</b>	0.824	0.815	0.816	<b>0.815</b>
<b>Fasttext</b>	0.835	0.828	0.827	<b>0.828</b>	0.825	0.828	0.825	<b>0.828</b>
<b>BERT</b>	0.841	0.832	0.834	<b>0.832</b>	0.843	0.838	0.839	<b>0.838</b>

FIGURE 6.2.9: MLCNN and SLCNN results for Adataset

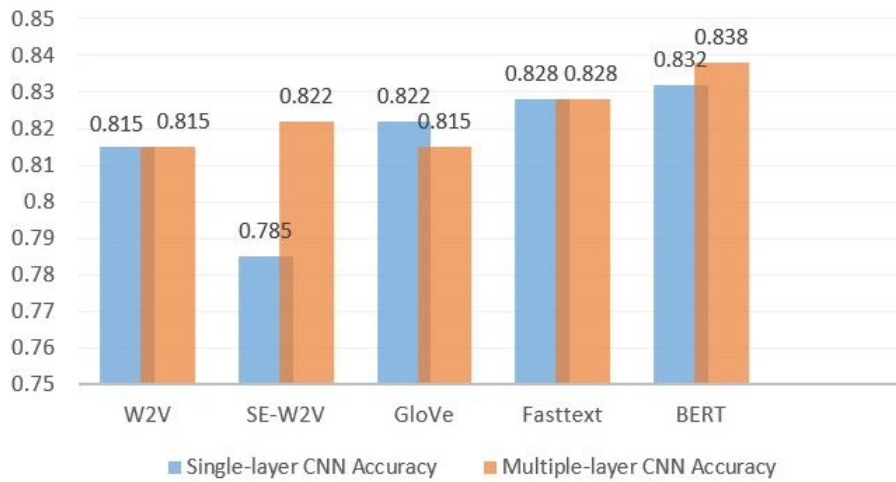


FIGURE 6.2.10: Chart MLCNN and SLCNN results for Adataset

### 6.2.2.2 Experiments set 8: Mdataset

The main results of these six experiments set are shown in table 6.8. These results show that the BERT model outperforms the TF-IDF and the other pre-trained models. Figure 6.2.11 shows the results of the MLCNN classifier with BERT model for each type of SATD categories.

	precision	recall	f1-score	support
REQ	0.812	0.517	0.632	151
DES	0.822	0.939	0.877	561
DEF	0.695	0.471	0.562	87
TEST	0.667	1.000	0.800	10
DOC	0.750	0.500	0.600	6
accuracy			0.809	815
macro avg	0.749	0.685	0.694	815
weighted avg	0.804	0.809	0.795	815

FIGURE 6.2.11: MLCNN and BERT performance metrics for each type of SATD in Mdataset

M Dataset						
Multiple-layer CNN						
	Precision	Recall	F1-score	Accuracy	Testing time	No Epoch
TF-IDF	0.721	0.748	0.707	<b>0.748</b>	0.148	7
W2V	0.779	0.793	0.776	<b>0.793</b>	0.206	7
SE-W2V	0.797	0.805	0.794	<b>0.805</b>	0.226	6
GloVe	0.792	0.799	0.784	<b>0.799</b>	0.218	7
Fasttext	0.783	0.791	0.783	<b>0.791</b>	0.228	6
<b>BERT</b>	0.804	0.809	0.795	<b>0.809</b>	0.316	10

TABLE 6.8: MLCNN performance metrics with all pre-trained models and TF-IDF for Mdataset

In these set of experiments, the results of MLCNN with Mdataset are improved in two models, BERT and SE-W2V. This result is the same result of MLCNN with Adataset. The accuracy of the other three models (W2V, Glove, Fasttext) is better with SLCNN for the two dataset. These results are tried to answer RQ4: "Does increasing the numbers of layers in CNN model improve the accuracy of the study approach?". The results of the two set of experiments (7,8) indicate that the single-layer CNN can perform better result with three pre-trained models, with no need to the deep in neural network. Additionally, the length of comments considered short as mentions in section 4.1.4. The length of comments range from 1 to 500, mostly falls between 1 and 50 words, and the average of numbers of words in the comment is 11. These results of the two experiments agree with the study [10] found that for text within 50 words a convolution pooling layer can be set, and within 500 words of text two convolution pooling layers can be used. The table 6.2.13 and the figure 6.2.14 show the results of MLCNN



and SLCNN for Mdataset. For the BERT model, the result is better when using MLCNN, because the architecture of BERT for word embedding is different from other models. For the SE-W2V, the main different between this model and the other models is the dimensional vector, SE-W2V is 200-dimensional and the other models is 300.

Figure 6.2.12 shows the multiple-layer CNN classifier with TF-IDF and pre-trained models.

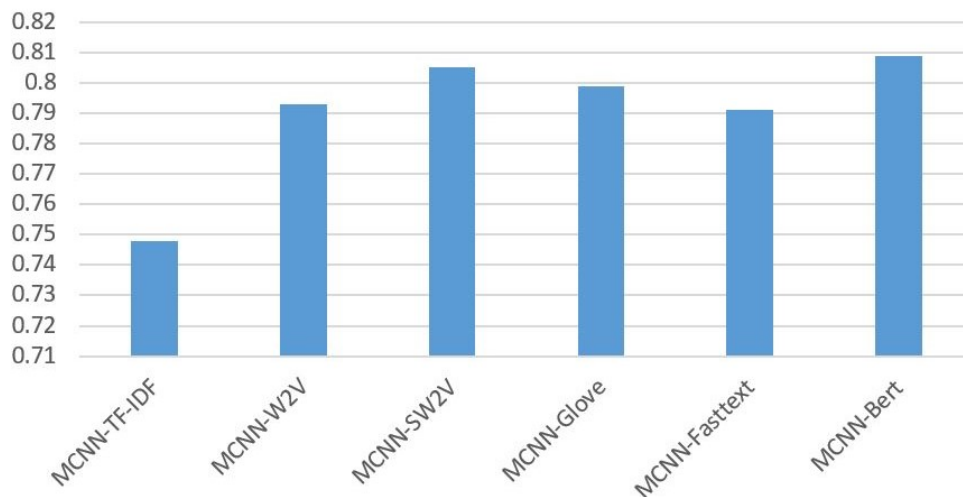


FIGURE 6.2.12: The accuracy of multiple-layer CNN with TF-IDF and pre-trained models for Mdataset.

	Single-layer CNN				Multiple-layer CNN			
	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy
W2V	0.807	0.812	0.798	<b>0.812</b>	0.779	0.793	0.776	<b>0.793</b>
SE-W2V	0.783	0.791	0.783	<b>0.791</b>	0.797	0.805	0.794	<b>0.805</b>
GloVe	0.802	0.802	0.785	<b>0.802</b>	0.792	0.799	0.784	<b>0.799</b>
Fasttext	0.804	0.806	0.788	<b>0.806</b>	0.783	0.791	0.783	<b>0.791</b>
BERT	0.796	0.804	0.792	<b>0.804</b>	0.804	0.809	0.795	<b>0.809</b>

FIGURE 6.2.13: MLCNN and SLCNN results for Mdataset

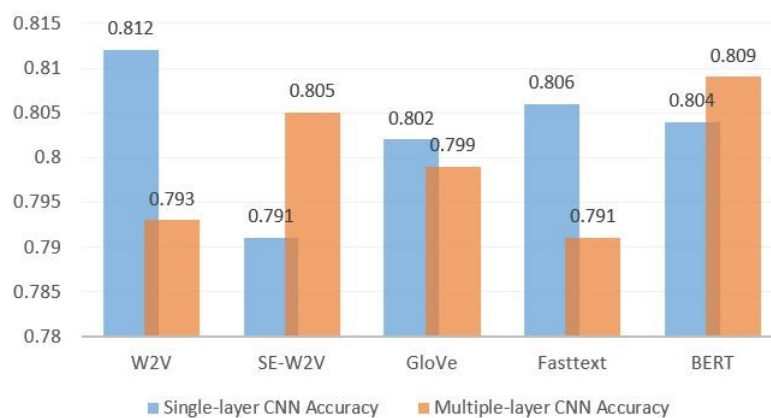


FIGURE 6.2.14: Chart MLCNN and SLCNN results for Mdataset

### 6.2.2.3 Experiments set 9: An exploratory experiment for combined-2 dataset

This experiment was conducted to answer the RQ3: "How well combined the two datasets improve classification accuracy?". In the two previous experiment sets 3 and 6, the accuracy of classifiers is not improved with the combined dataset. More analysis and experiments to explore the results of the best classifier in these two experiments (3,6). By comparing the f1-score for each type of SATD, the requirement and defect types are the loss accuracy. The first try was balance the dataset using oversampling by duplicate the minority classes (requirement, defect, test, documentation) to the majority class (design) using the re-sample from the sklearn toolkit<sup>1</sup>. Oversampling doesn't improve the accuracy of classifiers for the combined dataset. Second, the new strategy called exclude-one-merge-four was defined as shown in figure 6.2.15. This strategy repeated for each types of SATD with observation the accuracy until the requirement class excluded from Adataset and merge the other four classes of SATD,

<sup>1</sup><https://scikit-learn.org/stable/modules/generated/sklearn.utils.resample.html>

the accuracy is improved and outperforms the Adataset and the Mdataset. By repeating the same steps and exclude the requirement from Mdataset and merge the four types (design, defect, test, documentation), the result of accuracy was improved . Tables 6.9 , 6.10, 6.11 show the results of the combined dataset after excluding the requirement from the combining process. The explanation of these results refer the following reasons.

- Most of previous studies classify the SATD within one project, which some comments describe the code problem using the terms of specific domain, for example " // TODO should this use setDone()? // TODO should this use setFirst()?". This comments labeled as requirement TD that took from Mdataset. setDone and setFirst are the terms used in "apache-jmeter" project, not necessary use these terms in other project. This study used the comments and commits from seven projects of different domain, and combined these projects with 10 projects from Mdataset, and classified the SATD for cross-project. Additionally, When converting some terms to vectors using word embedding, the pre-trained model ignore some terms, because these terms did not exist in the vocabulary of model. For example the number of unique words after pre-processing the comments in Mdataset is 6108 words, by using word2vec pre-trained model to create words embedding dictionary, the model exclude 2011 words that are not exist in word2vec model.
- The requirement is the first stage in the software development process. It describes the functionality of the specific system considering the domain context of the system. By definition of requirement SATD, the comments

indicate that there is an ambiguous requirement that leads to the incompleteness of program, class or method [41]. This misunderstanding of requirement leads to design or defect problem, which leads to overlapping between these three classes. Let us demonstrate this by an example, the following comments are classified to the three classes (requirement, design, and defect).

- "// XXX add exceptions". From [apache-ant-1.7.0], classified as requirement (Mdataset).
- "TODO: We need a specific exception type here". from [argouml] classified as design (Mdataset).
- "// TODO: Shouldn't we throw an exception here?!?" from [argouml] classified as defect (Mdataset).
- "// XXX - should throw an exception instead?" classified as design (Mdataset).
- "TODO throw an exception since the query isn't valid?" from [WordPress] classified as defect (Adataset).

The term "exception" appears frequently in the four comments, which are classified into three different types of SATD. The term "throw" is frequent in two different types, and the number of neighbor words after pre-processing becomes small. As a result of that, the diversity of the projects increases these overlapping between comments.

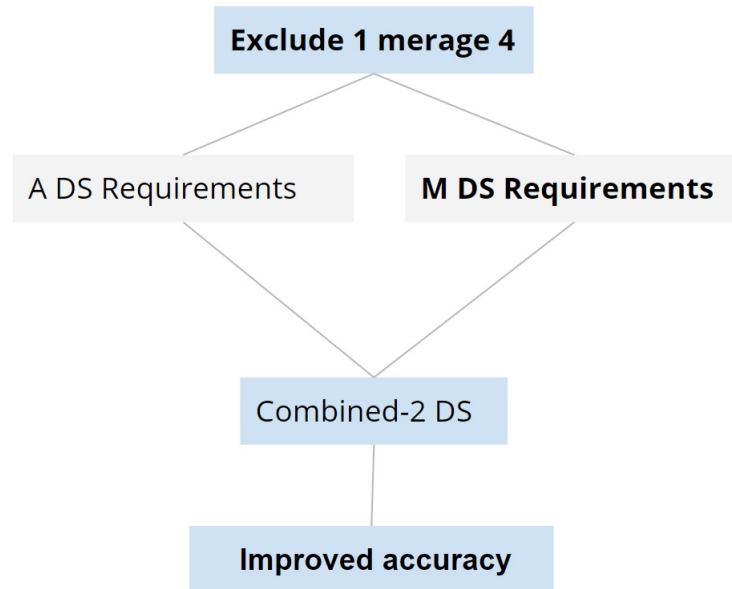


FIGURE 6.2.15: Exclude on merge four strategy

Combined-2 Dataset					
Naive Bayes					
	Precision	Recall	F1-score	Accuracy	Testing time
TF-IDF	0.645	0.705	0.609	<b>0.705</b>	0.016
Random Forest					
TF-IDF	0.82	0.826	0.807	<b>0.826</b>	0.053
Support Vector Machines					
TF-IDF	0.793	0.806	0.795	<b>0.806</b>	0.015

TABLE 6.9: ML classifier for combined-2 dataset after exclude requirements TD from Mdataset

Combined-2 Dataset						
Single-layer CNN						
	Precision	Recall	F1-score	Accuracy	Testing time	No Epoch
TF-IDF	0.791	0.801	0.785	<b>0.801</b>	0.125	6
W2V	0.82	0.824	0.799	<b>0.824</b>	0.086	7
SE-W2V	0.824	0.818	0.804	<b>0.818</b>	0.061	12
GloVe	0.827	0.822	0.799	<b>0.822</b>	0.066	6
Fasttext	0.828	0.829	0.805	<b>0.829</b>	0.109	9
BERT	0.829	0.839	0.834	<b>0.84</b>	0.148	10

TABLE 6.10: SLCNN for combined-2 dataset after exclude requirements TD from Mdataset

Combined-2 Dataset						
Multiple-layer CNN						
	Precision	Recall	F1-score	Accuracy	Testing time	No Epoch
TF-IDF	0.792	0.802	0.784	<b>0.802</b>	0.138	7
W2V	0.801	0.808	0.801	<b>0.808</b>	0.19	6
SE-W2V	0.816	0.818	0.803	<b>0.818</b>	0.095	7
GloVe	0.801	0.81	0.801	<b>0.81</b>	0.123	11
Fasttext	0.827	0.829	0.816	<b>0.829</b>	0.111	6
BERT	0.833	0.849	0.835	<b>0.849</b>	0.2	10

TABLE 6.11: MLCNN for combined-2 dataset after exclude requirements TD from Mdataset

Figure 6.2.16 shows all classifiers with TF-IDF, USE and pre-trained models.

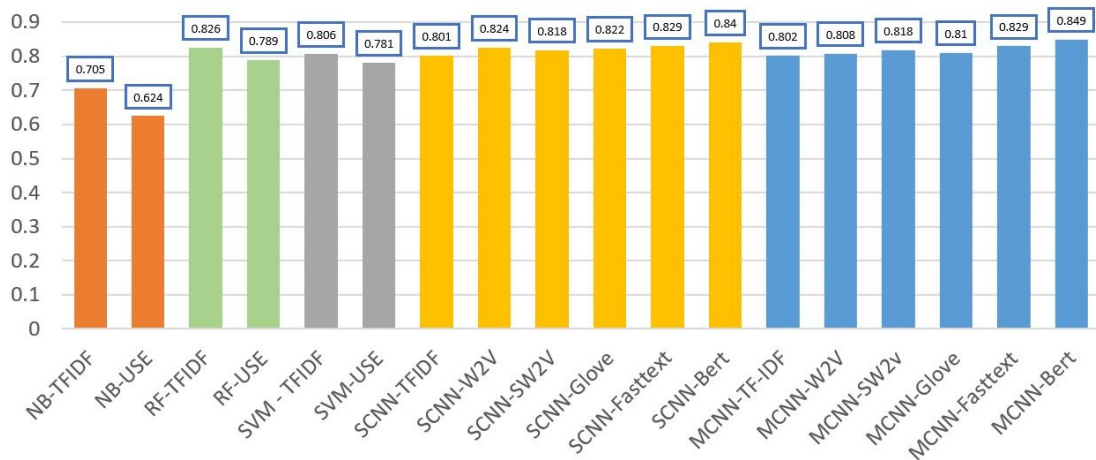


FIGURE 6.2.16: The accuracy of all classifiers with TF-IDF,USE and pre-trained models for combined-2 dataset.

### 6.3 Statistical Test

In all of the experiments, each experiment is repeated 10 times to estimate the variability of the results and to evaluation how close to each other. Also, to increase the accuracy of the estimate, assuming no bias or systematic error is present. The same classifier runs with four datasets, and the best performance is recorded for each one. To conduct suitable statistical test for the experiments study, the method that produced by [14] was adopted, which performs statistical test for multi classifiers over multiple datasets, that similar to this study.

This method compares the obtained results using Friedman non-parametric test. It ranks the classifiers for each dataset separately, then Friedman test compares the average ranks of classifiers over all dataset that are shown in table 6.12. If Friedman test finds statistically significant at  $p < 0.05$ , then null-hypothesis is rejected, it can proceed with a post-hoc test. The Nemenyi test is used to compare all classifiers to each other.

<b>Classifiers</b>	<b>Avg. Rank</b>
MCNN-BERT	4.37
RF-TFIDF	4.17
SCNN-BERT	4.11
SCNN-Fasttext	3.97
SCNN-W2V	3.77
SCNN-Glove	3.11
MCNN-SW2v	3.08
MCNN-Fasttext	3.01
MCNN-Glove	2.45
SCNN-SW2v	2.38
MCNN-W2V	2.38
SVM-TFIDF	2.19
RF-USE	1.92
MCNN-TFIDF	1.26
SCNN-TFIDF	1.19
SVM-USE	1.13
NB-TFIDF	0.46
NB-USE	0.33

TABLE 6.12: Average ranking sorted descending for all classifiers

The result of Friedman test for null-hypothesis: the distributions of all samples are equal was rejected with  $P=4.69E-09$ . Additionally, as the paper [14] recommended that Friedman chi-square is working better with  $N$  and  $k$  are big enough (as a rule of a thumb,  $N > 10$  and  $k > 5$ ) where  $N$  is the number of datasets, and  $K$  is the number of classifiers. The test was repeated with another non-parametric tests that should be preferred over the parametric ones



[14]. Kruskal-Wallis test used, and the P value was 1.50E-05. Since the Null hypothesis was reject. Then Nemenyi test conducted. Figure 6.3.1 shows the results of Nemenyi test, that present that there are statistically significant between accuracy of algorithms.

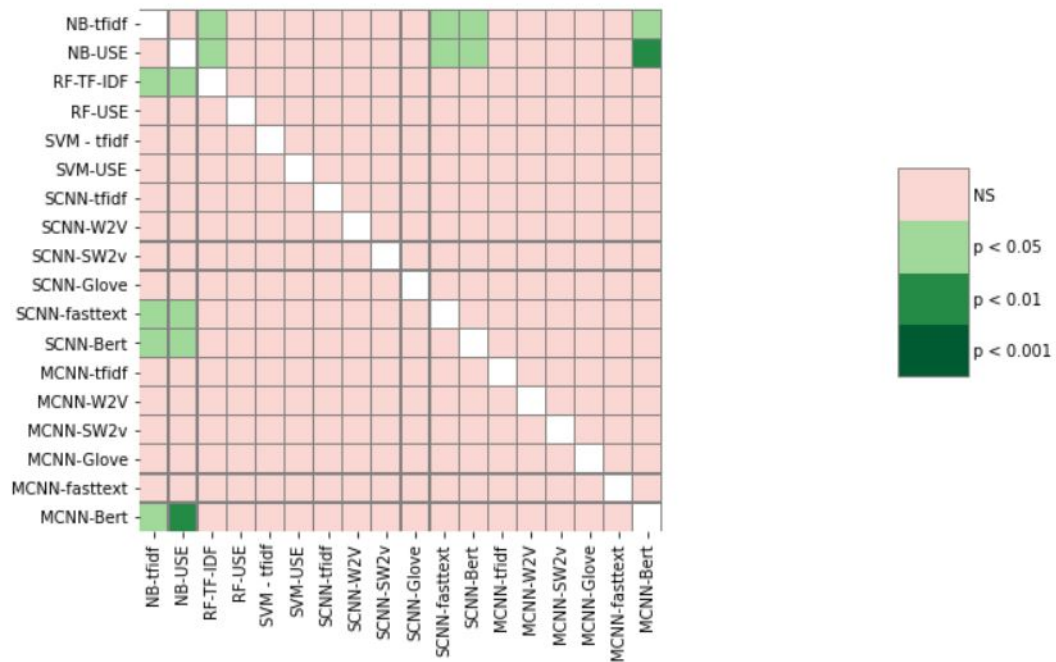


FIGURE 6.3.1: Nemenyi test Shows the p values for each pair

## 6.4 Discussion:

This section summarizes the discussion of results presented in the previous sections. Also, those results will be reflected on the research questions listed in the introduction and how the obtained results present answers to the questions. Let us start from the last experiment that answers the **RQ3: How well combined the two datasets improve classification accuracy ?**. For this question, three experiments 3, 6, and 9 were conducted with the combined datasets. In

experiment 3 the classical machine learning was used with three classifiers algorithms. The experiment was repeated by using CNN as a classifier with five pre-trained word embedding models. In these two experiments (3,6), the results of accuracy did not improve. So the exploratory experiment (9) perform to analyze the results. The main finding was the main reason that declined the accuracy overlapping between the comments, especially between requirement TD and design, that what happened during manual annotation for Adataset, most the difference between the first author annotation and the experts was in these two types of comments. In experiment (9) exclude-on-merge-four strategy was applied, and with excluding the requirement TD that belonged to Mdataset, and combined the other four types of SATD. As a result of this experiment, the accuracy was improved with the best classifier MLCNN and BERT model to 0.849.

For the first question **RQ1: How well the NLP pre-trained models can improve the identification of self-admitted technical debt from source code comments and commits effectively ?**. This study investigated the effectiveness of the NLP techniques for word representation and feature engineering to convert the words of comments to vectors. Six techniques used in all experiments can be the answer to **RQ1**. In experiments set 1 and 2, NLP count-based approach (TF-IDF) was used, and word embedding (USE) with three traditional machine learning algorithms (NB, RF, SVM). The results were the traditional ML improved the accuracy with TF-IDF comparing with ML and word embedding methods. In the experiments set 4 and 5, five types of word embedding methods were used, and TF-IDF with deep learning. The results of these experiments were; that deep learning working better with word embedding than TF-IDF. As a result that, NLP approaches improved the accuracy comparing with the baseline that used text-mining. In text-mining, the information could be patterns in

text or matching structure, but the semantics in the text is not considered. Additionally, for the two datasets (Adataset, combined-2 dataset), by using state-of-the-art NLP and deep learning, the accuracy is better than machine learning. The conclusion was the state-of-the-art NLP techniques, and the pre-trained models improve the accuracy of identification SATD. For the **RQ2:How well machine learning algorithms that include (SVM, NB, RF, and CNN) can automatically classify the five SATD types efficiently ?** To answer this question, this study investigated the effectiveness of the traditional and state-of-the-art techniques to classifying SATD. Three classical machine learning classifiers (NB,RF, and SVM) are used with TF-IDF and USE. The RF was the best classifier for three datasets with accuracy ( Adataset : 0.822, Mdataset : 0.820, and combined-2 dataset : 0.826). Moreover, the classic machine learning achieved the lowest results with USE; the result ranged from 0.726 to 0.795 for the three classifiers using Adataset, and from 0.639 to 0.807 for Mdataset. For deep learning and TF-IDF, the results ranged from 0.748 to 0.802 for all datasets. Using the state-of-the-art pre-trained models with deep learning, the results ranged from 0.791 to 0.849 for all datasets, and the best accuracy for BERT. The last **RQ4: Does increasing the number of layers in CNN model improve the accuracy of the study approach ?**. To answer this question, two types of CNN models (SLCNN, MLCNN) were used; the first model use the one embedding layer, one convolutional layer, and one pooling layer. In the second model, three layers for each type used. As the results show in experiments set (4,5,7,8,9), three models(W2V, Glove, Fasttext) working better with SLCNN. The SE-W2V was better with MLCNN for Adataset and Mdataset, and gave the same accuracy with combined-2 dataset. The BERT model was the best classifier with MLCNN in all datasets. As mentioned in experiments set 8, it has more than one factor to determine the best number of layers that will be used in the CNN model. The dataset size, the length of sentences, and the

architecture of word embedding model play a role in the number of layers for CNN model.

Figure 6.4.1 shows the accuracy of all the experiments results. The figures 6.4.2, 6.4.3 shows the confusion matrix for the two best classifiers with Adataset and Mdataset.

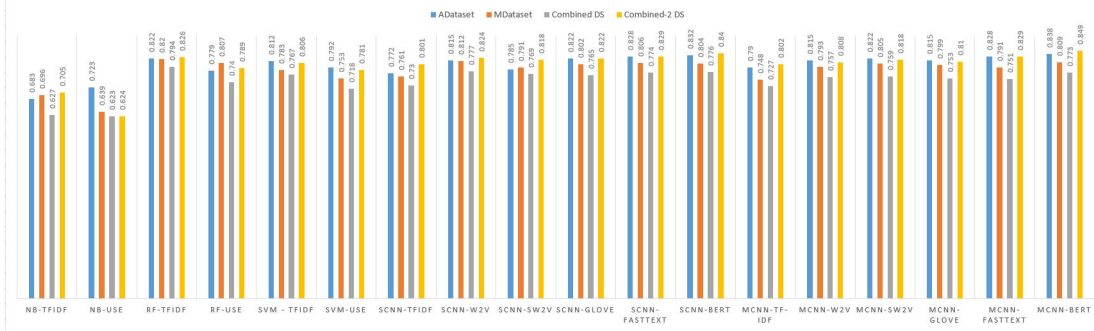


FIGURE 6.4.1

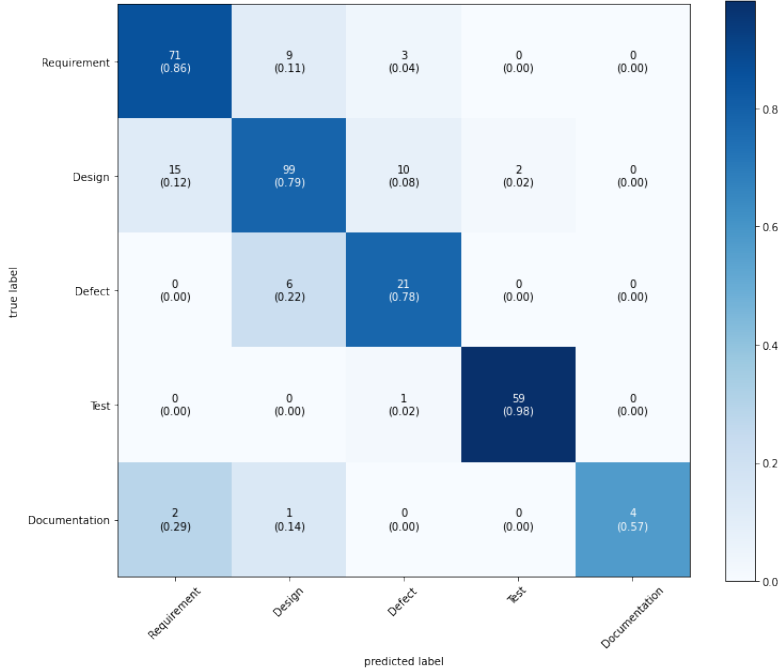


FIGURE 6.4.2: Confusion matrix for MLCNN with BERT model for Adataset

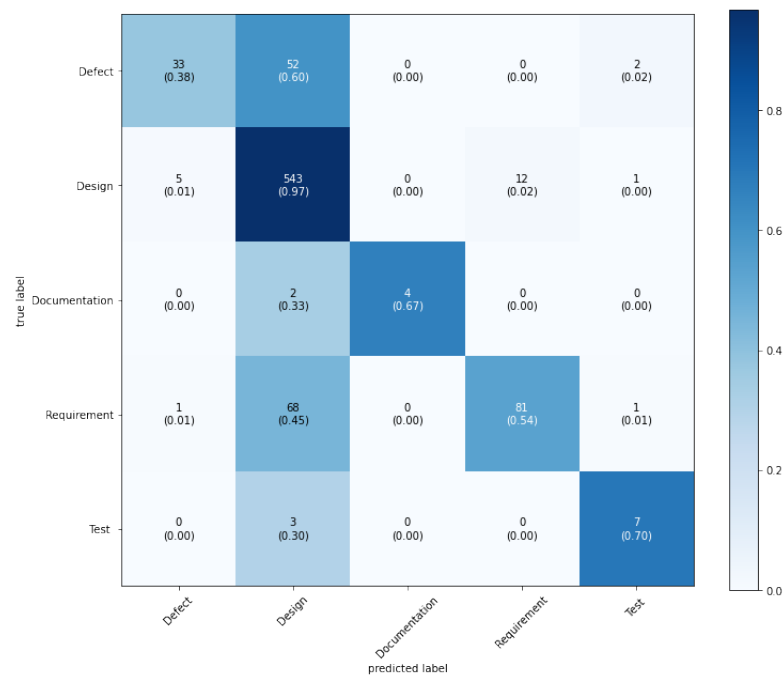


FIGURE 6.4.3: Confusion matrix for RF with TF-IDF for Mdataset

## Chapter 7

### Conclusion and future work:

Source code comments are explanation or annotation that written by the developers. Comments allow developers to clarify, document and express concerns about the implementation in an informal method that does not influence the program's functionality and are generally ignored by compilers and interpreters. Commits messages are the expression of action that the developers made on the source code and document this action with semantic commits. Self-Admitted Technical Debt (SATD) is comments or commits that indicate an issue in the part of code that needs additional cost. These issues are caused by a sub-optimal solution instead of using a better approach. There are five essential types of SATD that affect the software quality and should be identified. Manually analyzing SATD is tedious and time-consuming. This thesis presented classical machine learning and deep learning approaches to identify and classify SATD from comments and commits. Furthermore, This thesis investigated the effectiveness of NLP feature engineering techniques for SATD classification. Different NLP techniques were presented in this thesis, including TF-IDF and word embedding vectorization methods to feed in different classifiers.

Three datasets were used in this thesis; the first one is the public dataset

(Mdataset). This dataset consists of 62k comments extracted from 10 open-source projects, 4017 comments classified into five types of SATD, requirement, design, defect, test, and documentation. The second dataset (Adataset), that created by this study. Adataset included 5082 comments and commits from 7 open source projects that were classified as SATD. Manually labeled the SATD comments into five types of SATD, resulting in 1513 comments and commits manually labeled by the author and software experts. Kappa statistical test was applied to accept the label of each SATD comment to verify its authenticity. The test achieved a level of agreement measured between the author and experts +0.82 based on a sample including 0.17 of all technical debt types, which is considered almost perfect agreement according to Fleiss [18] values larger than +0.75 are characterized as excellent agreement. The last dataset is the combined two datasets.

A set of experiments are conducted for investigating the effectiveness of the NLP techniques and the effectiveness of the ML and DL techniques for identifying the SATD comments extracted from 17 open-source projects. The traditional well-known TF-IDF NLP techniques, and the state-of-the-art word embedding techniques USE, Word2vec, Golve, Fasttext, and BERT were used for representing comments into a numerical feature vectors. The TF-IDF depends on the statistical information of the comment sentence, whereas, the embedding methods focus on the semantic information of the sentences.

The evaluation performance method for this study will be by comparing the accuracy of classifiers for the three datasets. For classic machine learning, three types of the classifier were used (NB, RF, and SVM) with two types of word representation methods (TF-IDF and USE). The classical machine learning techniques are working better with traditional NLP techniques. This can be improved by comparing the best classifier accuracy (RF and TF-IDF) with (RF

and USE), the results for Adataset, with TF-IDF achieved accuracy 0.822, while USE 0.771. For Mdataset TF-IDF achieved 0.820, and USE 0.807. Finally, RF and TF-IDF with combined-2 dataset achieved accuracy 0.826. For deep learning approach, CNN classifier used with 5 NLP word embedding methods, and CNN with TF-IDF. The CNN and BERT model achieved best accuracy for Adataset: 0.838, and for combined-2 dataset: 0.849. The Word2Vec was the best according to Mdataset with accuracy 0.812.

### **7.0.1 Future work**

SATD is an essential indicator for evaluating software quality. This study approach reflects the effective way to identify the critical five types of SATD. In the future, The plan is to increase the scale of the proposed system by adopting more projects that developed in different programming languages, additionally in a different domain, for example, mobile applications, commercial software, medical domain. More investigation in others neural networks, deep learning architectures, and pre-trained models, by fine tuning the parameters of models in order to improve the accuracy of classifiers. Combined more than one classifiers, the main observation that some classifiers achieved f1-score better for one or two types of SATD. So the combination between classifiers might improve the accuracy of the model. For example, the classic machine learning could be used to classify the design and requirement and the deep learning to classify other types and combined them in one model.

Creating a specific pre-trained model for SATD by collecting a vast number of commits and commits from different software domains. At the same time, building this model using a different word embedding architecture such as BERT, GPT-3. Finally, the datasets of SATD need more analysis for more comments and commit, and using NLP techniques to draw the general pattern for



SATD.

## 7.0.2 Threats to validity

**Internal validity :** To classify the comments and commits for five types of SATD in Adataset. The threats to internal validity in this process include human factors to determining the correct identification of SATD types. To mitigate this threat, first, the comments and commits were collected depending on previous studies that classified these sentences to SATD. In the second step, all comments and commits that the first author classified was selected randomly through the website created for this job. After that, the comments that were classified randomly selected for the three experts to classifying again. Then, the level of agreement was evaluated between both experts and author by calculating Cohen's kappa coefficient [11]. The Cohen's Kappa coefficient is a widely used method to evaluate inter-rater agreement level for categorical scales, and it calculates the proportion of agreement that is chance-corrected. The result of the coefficient is scaled from -1 and +1, with a negative value indicating worse than chance agreement, zero means exactly chance agreement, and a positive value indicates better than chance agreement [17]. Whenever the value is closer to +1, the agreement is stronger. The level of agreement measured between the author and experts was achieved of +0.82 based on a sample including 0.17 of all technical debt types, which is considered almost perfect agreement according to Fleiss [18] values larger than +0.75 are characterized as excellent agreement.

**External validity :** To Considering the generalization of thesis findings. The public dataset that used in this study derived from comments of 10 open source project. To minimize external validity, new dataset was created from 7 open source project, two of them mobile applications. Additionally, the comments and commits were merged in the same dataset.

## Bibliography

- [1] Ravinder Ahuja et al. "The impact of features extraction on the sentiment analysis". In: *Procedia Computer Science* 152 (2019), pp. 341–348.
- [2] *AI Programming: 5 Most Popular AI Programming Languages* | Existek Blog. <https://existek.com/blog/ai-programming-and-ai-programming-languages/>. (Accessed on 04/10/2021).
- [3] Nicolli SR Alves, Thiago S Mendes, and G Manoel. "de Mendonça, Rodrigo O. Spinola, Forrest Shull, Carolyn Seaman, Identification and management of technical debt". In: *Information and Software Technology* 70 (), pp. 100–121.
- [4] Nicolli SR Alves et al. "Towards an ontology of terms on technical debt". In: *2014 Sixth International Workshop on Managing Technical Debt*. IEEE. 2014, pp. 1–7.
- [5] Venera Arnaoudova et al. "The use of text retrieval and natural language processing in software engineering". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. IEEE Computer Society. 2015, pp. 949–950.
- [6] Vimala Balakrishnan and Ethel Lloyd-Yemoh. "Stemming and lemmatization: a comparison of retrieval performances". In: (2014).

- [7] Gabriele Bavota and Barbara Russo. "A large-scale empirical study on self-admitted technical debt". In: *Proceedings of the 13th International Conference on Mining Software Repositories*. 2016, pp. 315–326.
- [8] David Binkley. "Source code analysis: A road map". In: *Future of Software Engineering (FOSE'07)*. IEEE. 2007, pp. 104–119.
- [9] Piotr Bojanowski et al. "Enriching word vectors with subword information". In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 135–146.
- [10] Peng Ce and Bao Tie. "An Analysis Method for Interpretability of CNN Text Classification Model". In: *Future Internet* 12.12 (2020), p. 228.
- [11] Jacob Cohen. "A coefficient of agreement for nominal scales". In: *Educational and psychological measurement* 20.1 (1960), pp. 37–46.
- [12] Ward Cunningham. "The WyCash portfolio management system". In: *ACM SIGPLAN OOPS Messenger* 4.2 (1992), pp. 29–30.
- [13] Robert Dale. "The commercial NLP landscape in 2017". In: *Natural Language Engineering* 23.4 (2017), pp. 641–647.
- [14] Janez Demšar. "Statistical comparisons of classifiers over multiple data sets". In: *The Journal of Machine Learning Research* 7 (2006), pp. 1–30.
- [15] Ai Deng. "Mining technical debt in commit messages and commit linked issues". PhD thesis. 2020.
- [16] Vasiliki Efstathiou, Christos Chatzilenas, and Diomidis Spinellis. "Word embeddings for the software engineering domain". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 38–41.

- [17] Joseph L Fleiss and Jacob Cohen. "The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability". In: *Educational and psychological measurement* 33.3 (1973), pp. 613–619.
- [18] Joseph L Fleiss, Bruce Levin, Myunghee Cho Paik, et al. "The measurement of interrater agreement". In: *Statistical methods for rates and proportions* 2.212-236 (1981), pp. 22–23.
- [19] Jernej Flisar and Vili Podgorelec. "Enhanced feature selection using word embeddings for self-admitted technical debt identification". In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2018, pp. 230–233.
- [20] Jernej Flisar and Vili Podgorelec. "Identification of self-admitted technical debt using enhanced feature selection based on word embedding". In: *IEEE Access* 7 (2019), pp. 106475–106494.
- [21] Martin Fowler. *Technical Debt Quadrant*. 2009. URL: <https://martinfowler.com/bliki/TechnicalDebtQuadrant.html> (visited on 10/13/2020).
- [22] Martin Fowler et al. "Refactoring: Improving the Design of Existing Code Addison-Wesley Professional". In: *Berkeley, CA, USA* (1999).
- [23] Mário André de Freitas Farias et al. "A contextualized vocabulary model for identifying technical debt on code comments". In: *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. IEEE. 2015, pp. 25–32.
- [24] Mário André de Freitas Farias et al. "Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary". In: *Information and Software Technology* 121 (2020), p. 106270.

- [25] Mário André de Freitas Farias et al. "Investigating the identification of technical debt through code comment analysis". In: *International Conference on Enterprise Information Systems*. Springer. 2016, pp. 284–309.
- [26] Krzysztof Gajowniczek, Arkadiusz Orłowski, and Tomasz Ząbkowski. "Simulation study on the application of the generalized entropy concept in artificial neural networks". In: *Entropy* 20.4 (2018), p. 249.
- [27] Yoav Goldberg. "A primer on neural network models for natural language processing". In: *Journal of Artificial Intelligence Research* 57 (2016), pp. 345–420.
- [28] Qiao Huang et al. "Identifying self-admitted technical debt in open source projects using text mining". In: *Empirical Software Engineering* 23.1 (2018), pp. 418–451.
- [29] P Huilgo. *Quick introduction to bag-of-words (BoW) and TF-IDF for creating features from text*. 2020.
- [30] Armand Joulin et al. "Bag of tricks for efficient text classification". In: *arXiv preprint arXiv:1607.01759* (2016).
- [31] A. Kedia and M. Rasu. *Hands-On Python Natural Language Processing: Explore tools and techniques to analyze and process text with a view to building real-world NLP applications*. Packt Publishing, 2020. ISBN: 9781838982584. URL: <https://books.google.ps/books?id=1AbuDwAAQBAJ>.
- [32] Yoon Kim. "Convolutional neural networks for sentence classification". In: *arXiv preprint arXiv:1408.5882* (2014).
- [33] Valentina Lenarduzzi, Alberto Sillitti, and Davide Taibi. "A survey on code analysis tools for software maintenance prediction". In: *International Conference in Software Engineering for Defence Applications*. Springer. 2018, pp. 165–175.

- [34] Valentina Lenarduzzi et al. "Are sonarqube rules inducing bugs?" In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2020, pp. 501–511.
- [35] Yikun Li, Mohamed Soliman, and Paris Avgeriou. "Identification and Remediation of Self-Admitted Technical Debt in Issue Trackers". In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2020, pp. 495–503.
- [36] Erin Lim, Nitin Taksande, and Carolyn Seaman. "A balancing act: What software practitioners have to say about technical debt". In: *IEEE software* 29.6 (2012), pp. 22–27.
- [37] Zhongxin Liu et al. "SATD Detector: A text-mining-based self-admitted technical debt detection tool". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*. 2018, pp. 9–12.
- [38] Long Ma and Yanqing Zhang. "Using Word2Vec to process big text data". In: *2015 IEEE International Conference on Big Data (Big Data)*. IEEE. 2015, pp. 2895–2897.
- [39] Aigars Mahinovs et al. *Text classification method review*. 2007.
- [40] Rungroj Maipradit et al. "Wait for it: identifying "On-Hold" self-admitted technical debt". In: *Empirical Software Engineering* 25.5 (2020), pp. 3770–3798.
- [41] Everton da S Maldonado and Emad Shihab. "Detecting and quantifying different types of self-admitted technical debt". In: *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. IEEE. 2015, pp. 9–15.
- [42] Everton da S Maldonado et al. "An empirical study on the removal of self-admitted technical debt". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2017, pp. 238–248.

- [43] Steve McConnell. *Managing Technical Debt*. 2008. URL: <http://www.construx.com/uploadedfiles/resources/whitepapers/Managing%20Technical%20Debt.pdf> (visited on 10/12/2020).
- [44] Tomas Mikolov et al. "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems* 26 (2013), pp. 3111–3119.
- [45] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).
- [46] monkeylearn. *Natural Language Processing (NLP) Guide – What Is NLP & How Does it Work?* 2020. URL: <https://monkeylearn.com/natural-language-processing/> (visited on 11/22/2020).
- [47] Robert L Nord et al. "In search of a metric for managing architectural technical debt". In: *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. IEEE. 2012, pp. 91–100.
- [48] Keiron O'Shea and Ryan Nash. "An introduction to convolutional neural networks". In: *arXiv preprint arXiv:1511.08458* (2015).
- [49] Data Monsters Olga Davydova. *Text Preprocessing in Python: Steps, Tools, and Examples*. 2018. URL: <https://medium.com/@datamonsters/text-preprocessing-in-python-steps-tools-and-examples-bf025f872908> (visited on 11/22/2020).
- [50] Sheetal S Pandya and NB Kalani. "Review on text sequence processing with use of different deep neural network model". In: *Int. J. of Advanced Trends in Computer Science and Engineering* 8.5 (2019), pp. 2224–2230.

- [51] Jeffrey Pennington, Richard Socher, and Christopher D Manning. “Glove: Global vectors for word representation”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.
- [52] Martin Porter. *The Porter Stemming Algorithm*. 2006. URL: <https://tartarus.org/martin/PorterStemmer/> (visited on 11/23/2020).
- [53] Aniket Potdar and Emad Shihab. “An exploratory study on self-admitted technical debt”. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE. 2014, pp. 91–100.
- [54] Leevi Rantala and Mika Mäntylä. “Predicting technical debt from commit contents: reproduction and extension with automated feature selection”. In: *Software Quality Journal* (2020), pp. 1–29.
- [55] Xiaoxue Ren et al. “Neural network-based detection of self-admitted technical debt: from performance to explainability”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28.3 (2019), pp. 1–45.
- [56] Rafael Meneses Santos, Methanias Colaço Rodrigues Junior, and Manoel Gomes de Mendonça Neto. “Self-Admitted Technical Debt classification using LSTM neural network”. In: *17th International Conference on Information Technology–New Generations (ITNG 2020)*. Springer. 2020, pp. 679–685.
- [57] Rafael Meneses Santos et al. “Long Term-short Memory Neural Networks and Word2vec for Self-admitted Technical Debt Detection.” In: *ICEIS (2)*. 2020, pp. 157–165.
- [58] Giancarlo Sierra, Emad Shihab, and Yasutaka Kamei. “A survey of self-admitted technical debt”. In: *Journal of Systems and Software* 152 (2019), pp. 70–82.



- [59] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. “Using natural language processing to automatically detect self-admitted technical debt”. In: *IEEE Transactions on Software Engineering* 43.11 (2017), pp. 1044–1062.
- [60] Behjat Soltanifar et al. “Software analytics in practice: a defect prediction model using code smells”. In: *Proceedings of the 20th International Database Engineering & Applications Symposium*. 2016, pp. 148–155.
- [61] Carmine Vassallo et al. “How developers engage with static analysis tools in different contexts”. In: *Empirical Software Engineering* 25.2 (2020), pp. 1419–1457.
- [62] Supatsara Wattanakriengkrai et al. “Automatic classifying self-admitted technical debt using n-gram IDF”. In: *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2019, pp. 316–322.
- [63] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. “Examining the impact of self-admitted technical debt on software quality”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 179–188.
- [64] *What is the best programming language for Machine Learning? | by Developer Economics | Towards Data Science*. <https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7>. (Accessed on 04/10/2021).
- [65] Wikipedia contributors. *Comment (computer programming)* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 18-December-2020]. 2020. URL: [https://en.wikipedia.org/w/index.php?title=Comment\\_\(computer\\_programming\)&oldid=993509204](https://en.wikipedia.org/w/index.php?title=Comment_(computer_programming)&oldid=993509204).

- [66] Baoxun Xu et al. "An Improved Random Forest Classifier for Text Categorization." In: *JCP* 7.12 (2012), pp. 2913–2920.
- [67] Aiko Yamashita and Leon Moonen. "Exploring the impact of inter-smell relations on software maintainability: An empirical study". In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 682–691.
- [68] Meng Yan et al. "Automating change-level self-admitted technical debt determination". In: *IEEE Transactions on Software Engineering* 45.12 (2018), pp. 1211–1229.
- [69] Zhe Yu et al. "Identifying Self-Admitted Technical Debts with Jitterbug: A Two-step Approach". In: *arXiv preprint arXiv:2002.11049* (2020).
- [70] Nico Zazworka et al. "Comparing four approaches for technical debt identification". In: *Software Quality Journal* 22.3 (2014), pp. 403–426.

## Chapter 8

### Appendix A

#### 8.1 Database ER-Diagram

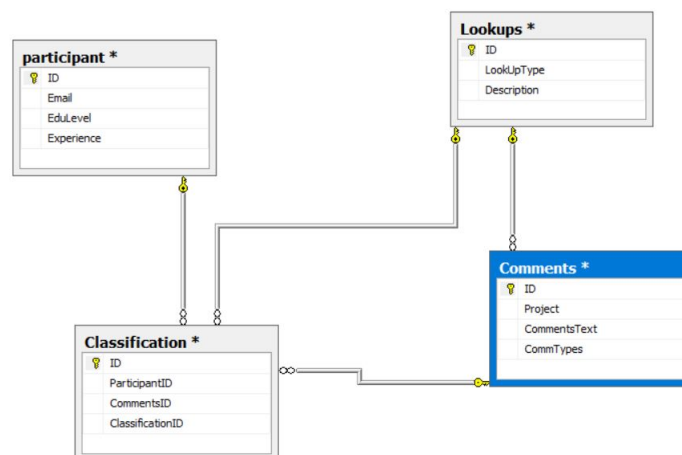



FIGURE 8.1.1: Database ER-Diagram

#### 8.2 Website Pages

Home The Task Contact

  
 BIRZEIT UNIVERSITY

## Labeling Self-Admitted Technical Debt Comments and Commits

To complete the requirement for the Master of Software Engineering program at Birzeit University, these classifications are part of the master's thesis preparation, so please help with this work.

Gratefully

**Supervisor:** Dr. Abuaseoud Hanani

**Student name:** Ahmed Sabbah

### Your task

1. After carefully reading the text below, you should select what type of self-admitted technical debt that should be classified as.
2. Choose one of the six labeled from List in the screen (i.e, Design, Defect, Documentation, Requirement, Test or Not technical debt )
3. Please be careful, all comments and commits that will presented are classified as technical debt without type by using tools and other studies.
4. There is no time limit to complete your task, every time you need to repeat the task you will get new instance
5. To avoid frequently present the same comments, you will enter the valid email address
6. You can start here :

[Start task](#)

FIGURE 8.2.1: Home Page part 1

### Source code comments

Source code comments are explanation or annotation that written by the developers, comments allow developers to clarify, document, and express concerns about the implementation in an informal method that does not influence the functionality of the program, and are generally ignored by compilers and interpreters..

### Commits messages

Commits messages are the express of action that the developers made on the source code and document this action with semantic commits

### Technical debt

is a metaphor, coined by Ward Cunningham . It reflects the additional cost that imply to rework caused by a sup-optimal solution instead of using the better approach in software development life cycle. The concept of TD is derived from financial debt, as the Interest resulting from the late payment. Similar to the financial dept, TD has an interest and the cost increases if not pay the debt early, by refactoring the code on the suitable time, to avoid interest in the future.

### Self-Admitted technical debt - SATD

is a technical debt that written by the developers deliberately, through comments or commits messages , with the knowledge that the implementation is not an optimal solution for the software.

**Self-admitted design debt:** These comments indicate that there is a problem with the design of the code. They can be comments about misplaced code, lack of abstraction, long methods, poor implementation, workarounds or a temporary solution. The following source code comments are examples of self-admitted design debt:

- // PR: I do not know what to do if the object class // has multiple defines // but this is for logging only... -[from apache-ant-1.7.0]
- // I hate this so much even before I start writing it. // Re-initializing a global in a place where no-one will see it just // feels wrong. Oh well, here goes." - [from ArgoUml]
- TODO: - This method is too complex, lets break it up - [from ArgoUml]
- TODO: really should be a separate class - [from ArgoUml]
- TODO: move this to components -- the only reason why it's in core is because // it's used as a guinea pig by a couple of tests -[from apache-jmeter-2.10]
- // XXXX re-evaluate this //can getSuper work by itself now? //if we're a class instance and the parent is also a class instance //then super means our parent.-[from]Edit-4.2 ]

FIGURE 8.2.2: Home Page part 2

**Self-admitted defect debt:** In defect debt comments the author states that a part of the code does not have the expected behavior, meaning that there is a defect in the code.

- `// Bug in above method" - [from Apache Jmeter]`
- `// WARNING: the OutputStream version of this doesn't work! - [from ArgoUml]`
- `// FIXME formatters are not thread-safe-[from apache-ant-1.7.0]`
- `// TODO: Something might go wrong during processing. We don't really // want to create the model element until the user releases the mouse // in the place expected,[from-argouml]`
- `// todo: is this comment still relevant ?? // FIXME: need to use a SAXSource as the source for the transform // so we can plug in our own entity resolver-[from apache-ant-1.7.0]`

**Self-admitted test debt:** Need for Implementation or Improvement of the current tests. Lack of tests, inadequate test coverage, and improper test design

- `// TODO should this be done even if not a full test plan? // and what if load fails?-[from apache-jmeter-2.10]`
- `// not sure whether this test is needed but cost nothing to put. // hope it will be reviewed by anybody competent-[from apache-ant-1.7.0]`
- `// cleanAllExtendsBut(model); // TODO: why is this causing a crash?-[from ArgoUml]`
- `// TODO: Test Mac keyboard accelerator changes done here by mivingstone // shortcut key`

**Self-admitted requirement debt:** Comments indicate that there is an ambiguous requirement that leads to incompleteness of the method, class or program.

- `// TODO no methods yet for getClassname" - [from Apache Ant]`
- `// TODO no method for newInstance using a reverse-classloader" - [from Apache Ant]`
- `// TODO: The copy function is not yet * completely implemented - so we will * have some exceptions here and there."/ - [from ArgoUml]`
- `// TODO support: multiple signers -[from apache-jmeter-2.10]`
- `// Set the overall status for the transaction sample // TODO: improve, e.g. by adding counts to the SampleResult class-[from apache-jmeter-2.10]`

**Self-admitted documentation debt:** Incomplete comments, lack of code comments, no documentation for important concerns, poor documentation, author express that there is no proper documentation supporting that part of the program.

- `• **FIXME** This function needs documentation-[from columba-1.4-src]`
- `// FIXME: Document difference between warn and warning (or rename one better)-[from jrubby-1.4.0]`
- `// TODO Document the reason for this" - [from Apache Jmeter]`
- `// Fixme: Only F_SETFL and F_GETFL is current supported // FIXME: Only NONBLOCK flag is supported // FIXME: F_SETFL and F_SETFD are treated as the same thing here. For the case of dup(fd) we // should actually have F_SETFL only affect one (It is unclear how well we do, but this TODO // is here to at least document that we might need to do more work here. Mostly SETFL is // for mode changes which should persist across fork() boundaries. Since JVM has no fork // this is not a problem for us.`

**For more details you can refer to :**

- *E. d. S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical Debt," 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), Bremen, 2015, pp. 9-15, doi: 10.1109/MTD.2015.7332619.*
- *N.S.R. Alves, T.S. Mendes, M. G. de Mendonca, R.O. Spinoia, F. Shull, and C. Seaman. Identification and management of technical debt: A systematic mapping study, Information and Software Technology, 70:100–121, 2016.*

FIGURE 8.2.3: Home Page part 3

Home
The Task
Contact

## Labeling Self-Admitted Technical Debt Comments and Commits

Email	Job Title/Description	Experience years	Action
<input type="text"/>	-- Select --	-- Select --	<input type="button" value="Next Step"/>

**Notes:**

If you have participated before, please enter the same email address in order to continue the classification.

FIGURE 8.2.4: Information of participant

Participant Email: <a href="mailto:asabbah44@gmail.com">asabbah44@gmail.com</a>		Participant Comments classified : 0	
Comment/Commit	Classification	Action	
moved some code within the activation area into new methods. added the ChainsawCentral as a hard coded plugin for now.	-- Select --	<input type="button" value="Save"/>	<input type="button" value="Skip"/>

**Notes:**

- If the Comments/Commits text not appear, please click on the skip, to get a new one.
- If you are not certain of the comments type, you can skip it.
- Some **Commits** indicates that related or solved type of debt. Choose the type of debt. E.g.
  1. Add some explanations about what's going on--> **Documentation**
  2. Added unit test to show the issue--> **Test**
  3. The only feature which we don't support is correlated message groups. That requires a bit more work and also may complicated the configuration a bit as you would need to configure- expression to evaluate the correlation keyAnd then the throttler logic need to have a map of correlation key --> **Requirement**
- Please try to classify as you can, and try to not less than 20 comments.

**Definition:**

- **Design debt:** Indicate that there is a problem with the design of the code. They can be comments about misplaced code, lack of abstraction, long methods, poor implementation,workarounds or a temporary solution. **Eg.** //TODO This method is too complex
- **Defect debt:** In defect debt comments the author states that a part of the code does not have the expected behavior, meaning that there is a defect in the code. **Eg.** // Bug in above method
- **Requirement debt:** Comments indicate that there is an ambiguous requirement that leads to incompleteness of the method, class or program.**Eg.** // TODO support multiple signers
- **Test debt:** Need for implementation or improvement of the current tests. Lack of tests, inadequate test coverage, and improper test design. **Eg.**//TODO - need a lot more tests
- **documentation debt:** Incomplete comments, lack of code comments, no documentation for important concerns, poor documentation, author express that there is no proper documentation supporting that part of the program. **Eg.** // TODO Document the reason for this

FIGURE 8.2.5: Classification page