Self-admitted Technical Debt Detection Using NLP

CSCI 6509 - Natural Language Processing

February 7, 2020

Project ID: P-10

Jadeja Nirav - B00789139
Patel Supriya - B00791627
Singh Abhishek - B00782673

Contents

List of Figures

List of Tables

1 Abstract

The term Technical Debt (TD) has been defined in many different ways but in the simple form, we define it to be the shortcuts taken by developers to meet deadlines or the hacks they perform for the time being to be the solitary one in the market. From the literature, it was perceived that approximately 75% of developers were not even familiar with the term though they admitted to performing Technical Debt on a regular basis when explained a generic definition. The concern with this is that there are so many forms of TD that even developers may not be aware of and this only leads to accumulation of refactoring work in the future. Among the several types of TD such as Design, Architectural, Bit Rot, Accidental, Requirement, etc. we aim to focus on design and requirement debt. TD due to requirement debt typically occurs when developers do not understand the requirements accurately and instead of clarifying them, they make assumptions which in turn lead to additional work in future. Mistakes like these can be overcome easily if detected on time. Moreover, from time to time the structure of the code is defined as unstable or fragile which happens when developers try to fix the system for time being and this approach has also been termed as "Frankenstein" approach by many. Hence, the problem we aim to tackle in this project is the detection of requirement and design TD that is Self-Admitted by developers themselves (i.e. not accidental) which would, in turn, prevent future costs and time, not to mention increased efforts as well.

2 Introduction

Our project is an implementation of an analogous research paper we liked [1]. We focus on the detection and understanding of the concept of Technical Debt. We are interested in this project because the issue it presents is ubiquitous but not many realize its significance and the effect it has on the overall program/project. It can start being as small as not naming variables properly and adding a comment regarding its functionality which in future will become problematic if someone else misunderstands and either removes an important functionality or changes it. In this case, if a bug is encountered it will become difficult to understand the source as the code would be correct but missing functionality which no one is aware of any more. TD is precisely the debt which is anonymously accumulated overtime and creates unfathomable difficulties.

When conducting interviews with 35 practitioners, about 26 practitioners were not familiar with the term but when explained, were familiar with its implication. Majority of them accepted that quite a lot of decisions are made instantly due to pressure from customers but it later on results in travelling to customer sites in some cases to solve issues with the program which in turn is more expensive and time-consuming. Sometimes if the pressure is too much then assumptions are made without clarifying all the requirements resulting into requirement debts. They discussed a few strategies for TD where one participant suggested an approach to allocating some percentage of each release cycle to deal with TD. One of the solutions is to communicate with customer and explain the consequences and trade-offs while many may still ignore the term until the system finally starts fracturing [2].

The other thing with TD is that not everyone is aware of its various types and their distinct effects. There are two main debts, one where the person introducing it is not aware and the second is Self-Admitted Technical Debt (SATD) where the individual is aware of the shortcuts taken and its consequences in future. These two classes cover a vast amount of debts under them. According to us, the most basic type of TD starts at People Debt where the development activity is delayed, and the process hindered because of the absence of the required amount of people on the project due to improper hiring and training activities. This debt can actually escalate all other debts because not having a good team or sufficient people on the team will affect the overall pressure an individual goes through when having to meet a deadline. The second debt is documentation debt

which is present due to incomplete documentation. Even though a project is functioning perfectly in the current situation, but an absence of proper documentation affects its future progress and in turn costs time and expense. Some of the most imperative debts are design, requirement, architecture and test debt. They occur when one chooses bad coding practice, for example, classes or structures that occupy unnecessary space and consume more processing power. When requirements of projects are not clarified properly before implementation leads to the fragility in the program. Sometimes, security is not implemented globally but instead to some sections, or trade-offs are made when facing some issues, which also leads to requirement debt. In case of low performance and robustness or improper modularity of the code, we refer to the debt as architecture debt. At the time of creating test cases, if a sufficient amount of code is not covered and failed test cases are not acted upon then it creates test debt [3].

From the above-mentioned TDs, we aim the focus of our project on SATD caused by design and requirement debts. These debts are expressed more openly in code comments and through them, we aim to detect the technical debt. After parsing the code comments, we observe certain keywords dedicated to a particular category of TD and after training a model we will be able to detect not only TD but what type of TD exists as well. To test this theory, ten open source projects have been selected and they vary from each other besides having a different amount of source code comments present. Currently, we are using a dataset which is already available to us, but it is not processed. The biggest advantage we have is that it is already manually classified which saved us considerable time. This manual classification was created as a part of a project where it consumed a total of 185 hours and interviews were taken of the participants whose data once collected was checked for Cohen's Kappa coefficient. This coefficient is used to validate the response of the participants and measure how much percentage of them agree on the classification. It is very essential to measure this because the definition of TD differs from person to person as not everyone is too intent on recognizing it. The level of agreement is found to be $+0.81$. After the extraction, we need to apply several filters to remove special characters and source code which have been commented. At the same time, we remove license comments and convert the entire text to lowercase. Once we have processed data, we train our model and test it on a test dataset. Cross-validation is applied to make sure that our model does not overfit. We will be measuring the efficiency of our model by comparing the F1 score with some baselines from the literature review. The report hereafter has a brief literature review highlighting some of the works into this field followed by the

methodology we adapted, and the results explained in the next section. We present a critical review and our future goals in the form of conclusion followed by references and additional details in the appendix.

3 Related Work

Technical debt has been around for a long time, but the term was officially recognized by Cunningham. Figuratively, it has been called something which is incomplete or inadequate and in turn affects the development lifecycle in terms of quality and costs in the future. Authors of one of the first paper [1] gave a brief introduction concerning the term and proposed a framework which would validate the theory regarding the relationship between the cost and necessity of TD. They take into account some phenomena such as document missing, incomplete tasks mounting, misunderstanding the requirement, write only codes, etc. and propose an interesting approach for the management of technical debt [4]. To have a better understanding of the term, we wanted to know the point of views from software practitioners as well [2]. To understand how developers dealt with the TD, the authors conducted an interview with 35 participants from different places and created a generic questionnaire to get an overall idea regarding their experience in the field, how old the term is, was it necessary and if yes then how it is explained to the customer? From these questions, they were able to get an outline where they observed that 75 percent of people were not aware of the term but when explained its meaning, one of them quoted, "Familiar with it? We live with it every day." They obtained a rather diverse review concerning its inevitability. Even when they admitted to incurring technical debt themselves due to deadline pressures, they confined that it, in turn, causes increased cost and time consumption not to mention substandard code quality. Sometimes, it is accommodating because by being the first one in the market, you can also assemble customer feedback which in turn you can use it to your advantage by building something that customer desires.

The struggle is the identification of TD. Next, we reviewed a paper where they studied TD and its types [5]. The primary aim was to understand how much actual TD is reported by analysis tools used when compared to manually classified data in a project. One of the speculations in the paper was whether manually classified TD can be used to make a better development tool which would be more efficient, and this led us to contemplate if the analysis tools are efficient at all. An interesting observation from the paper is that intentional debt is present in almost every category. Another concern is the method used for detection. Source code analysis may not be the best approach. A fairly nice paper which we came across is on the systematic mapping of the literature and it covers abundant information regarding the TD. There is also brief research into techniques to

identify and manage TD [6]. In all the previous literatures, technical debt had been generally defined but [3] covers almost all the types of technical debts present such as test debt which occurs when the tester does not include all the scenarios, documentation debt due to incomplete information, architecture debt when the code is not modular and is written in a poor way, etc. The review of ontology terms is really interesting because it brings all the definitions under one roof which helps everybody understand the elementary features of various types of debt. Out of all the above debts, design debt is more significant. There have been several attempts in the literature to detect and manage design debt. One of them is where the authors created a framework which selects design flaws and defines rules for each of them to see their effect on code whether it is positive or negative. Although this seems competent, one disadvantage is that it is not generalised [7]. One of the most common approaches towards the maintenance of TD is refactoring the code, i.e. changing the structure of the code without changing its functionality. This is mainly done when the current structure cannot handle the requirements of the clients [8]. From all this it is evident that most of the debts are self-admitted, meaning developers are aware that they are producing debts. Hence, SATDs are relatively easier to detect and more important to manage. It was done so by the authors where they initially detected the SATDs in source code comments. Commenting code is a good habit and a well-commented code can give us so much information to tie the case up. These comments even though they are from different users have some similar characteristics through which they were able to understand it. They have also found the relativity between the debt and time or complexity of the code [1]. It becomes necessary to study the effect of SATD on software quality [9]. This paper reviews the effect of SATD on quality by studying the effect in code having SATD and comparing it with the ones without it and whether the changes needed to be done become difficult with the presence of SATD or not. To measure the defects in a code, they first look into all the changes that have been made in the code till date by learning through keywords such as, "fixed issue ID", "bug ID", "patch", etc. In order to see if the defect is fixed, they use regular expressions to extract logs. From these logs, they are able to make out the reason behind the defect by checking when the issue was fixed and what caused it in the first place.

The next step to understanding the technical debt is working on how to manage it. In a review, it had been stated that static analysis tools are not effective when it comes to detection of SATD, most of the times these tools fail [10]. But on the other hand, several papers [11][9][12], use Natural Language Processing for their purpose. NLP al-

lows parsing of text directly which becomes advantageous because the source code comments carry maximum information regarding the SATD. NLP provides functionalities for pre-processing of the obtained dataset in the form of stop words removal, tokenizing, stemming, and vector space calculation. Also, the regular expressions help in detection and removal of comments which provide no information regarding SATD such as license code comments. Our project implementation has been based on a research paper by Maldonado, Shihab and Tsantalis [1], where we aim at detection of SATD using NLP. For this, we require source code comments which we extract for ten open source java based projects. This is done by a java plugin known as JDeodorant. It is mainly used for type checking in Object-Oriented programming language as we saw that not always the principle is followed, and refactoring is required [13]. It proves to be quite an efficient tool as with its help it becomes possible to detect TD and also at which line in the code it is along with the type of comment. From the literature review, we present the methodology we use for our work in the following sections.

4 Problem Definition and Methodology

4.1 Research Problem

As we know, in a general way we can define technical debt as shortcuts taken by a developer to deliver features under a given time frame. Further in future, it may result in bad quality of the software. One way to detect technical debt is source code analysis. Mostly, it relies on syntax tree or any other advanced source code representations. Nowadays, code smell detectors also generate possible ways to resolve detected debt [14][15]. But these solutions are computationally expensive.

Another way to tackle this problem is to rely on source code comments for further detecting technical debts. There are several benefits using this approach over traditional. First, it is not computationally expensive compared to source code analysis. As source code comments can be extracted easily using regular expressions. Second, it does not depend on threshold values which are required in all metric based code smell detection approaches. As finalizing certain values for this approach is a challenge.

4.2 Methodology

- **Language:** Python
- **Tools & Library:** Sklearn, Keras, Pandas, Numpy, NLTK, Matplotlib, Wordcloud

We have taken dataset [1] which was published by Everton da S. Maldonado, Emad Shihab and Nikolaos Tsantails. The reason behind using dataset rather than deriving new one from source code is that it is well classified. They extracted source comments from popular 10 open-source Java-based projects as mentioned below:

1. **Ant:** A build tool
2. **ArgoUML:** UML modelling tool
3. **Columba:** An email client
4. **EMF:** The modelling framework and code generation facility for building tools and applications
5. **Hibernate:** It provides Object Relational Mapping (ORM) support to applications

6. **JEdit:** Text editor
7. **JFreeChart:** Chart library
8. **JMeter:** Load testing tool for analyzing and measuring performance of various services.
9. **JRuby:** Pure-Java Implementation of Ruby programming language
10. **SQuirrel:** A graphical SQL client written in Java

Once the source code is obtained from all projects, the authors have used JDeodorant [13], an open-source Eclipse plug-in to extracts the comments. As it provides detailed information about source code comments such as type (i.e. Block, Javadoc, Single line), location (the i.e. line where they start and end) and their context (i.e. method, type and field where they belong to). Further, they applied several filtering operations: ignoring license comments, commented code, and Javadoc. At last, authors have classified comments into design, defect, test, implementation, documentation and without classification category [1]. Original dataset can be found at 'Dataset/technical_debt_dataset.csv' in project repo. As further details can be seen in the below tables [1] and pictures.

Project	**Project Details**			**Comment Details**	
	Release	**# of classes**	**# of Contributors**	**# of comments after filtering**	**# of TD comments**
Ant	1.7.0	1475	74	4137	131
ArgoUML	0.34	2609	87	9548	1413
Columba	1.4	1711	9	6478	204
EMF	2.4.1	1458	30	4401	104
Hibernate	3.3.2 GA	1356	226	2968	472
JEdit	4.2	800	57	10322	256
JFreeChart	1.0.19	1065	19	4423	209
JMeter	2.10	1181	33	8162	374
JRuby	1.4.0	1486	328	4897	622
SQuirrel	3.0.3	3108	46	7230	286
Average		1625	91	6257	407
Total		16249	909	62566	4071

Table 1: Details of dataset

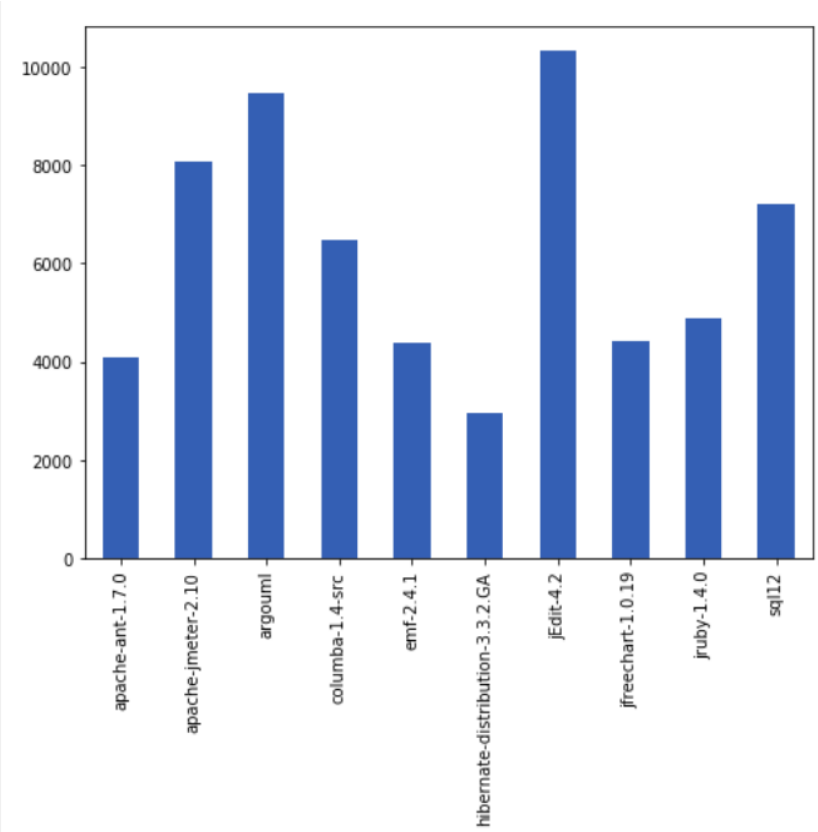

Figure 1: Number of comments for selected projects

Debt type	**# of comments after filtering**
Defect	472
Design	2703
Documentation	54
Implementation	757
Test	85
Without Classification	58204

Table 2: Number of comments after filtering corresponding to debt type

Once, we got our hands-on dataset, we observed that several modifications must be made for further cleaning. First, we removed all the comments data which wasn't classified into any category. The previous operation resulted in reducing data size from over 60,000 rows to 4071 rows. Then, we considered 'defect' and 'implementation' type of debt as 'requirement' debt. Further, the remaining 2 types of debt 'documentation' and 'test' grouped and categorized as 'other' type debt.

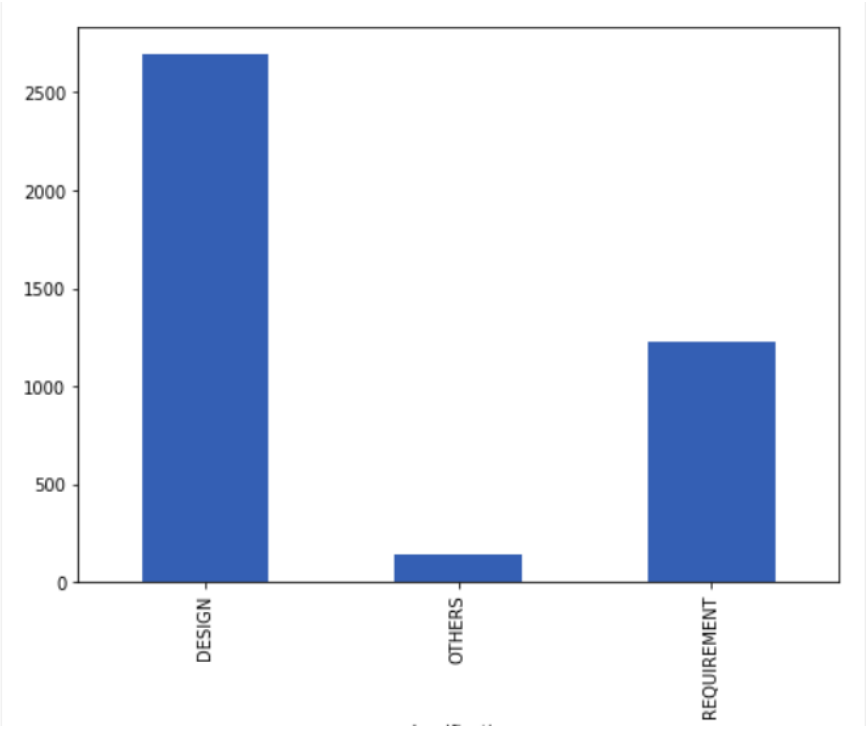

Figure 2: Design, requirement and other debt after filtering

For further cleaning, we removed all special characters from the processed dataset except '!' and '?'. We considered these 2 special characters as information in our case as it helps to distinguish between types of debts. We also eliminated HTML tags and other code snippets inside comments (i.e. ThisMethod(); takes too many arguments). Here, we utilized regular expressions for cleaning operations as mentioned above. At last, we applied to stop words removal from the NLTK library. At this point, we didn't apply stemming operation because it was modifying critical words (i.e. 'fixme' to 'fixm'), but we implemented stemming operation and placed it inside comment for future usage. The processed dataset can be found at 'Dataset/processedDataset.csv'.

Below, there are definitions and comments for design and requirement debt to provide a broader view.

4.2.1 Self-admitted design debt

Usually, this type of debt represents the problem within the design of code. Prime reasons for occurrences are long methods, temporary solution, inadequate abstraction and comments related to misplaced code. It can be resolved via refactoring or by rewriting code to achieve speed, security and so forth.

"TODO:- This method is too complex, lets break it up" - [ArgoUML]

"//quick dirty, to make nested mapped p-sets work:" - [Apache Ant]

As from the above examples, we can say that code authors are determined to refactor the code where it is not the idle solution. Hence, we can keep it under design debt category. Here, design debt considered based on the refactoring current solution.

4.2.2 Self-admitted requirement debt

Requirement type debt stands where all requirements for the implementation are not met yet as can be found in the below examples.

*"TODO: The copy function is not yet * completely implemented - so we will * have some exceptions here and there.*/"* - [ArgoUML]

"TODO: This dialect is not yet complete. Need to provide implementations wherever Not yet implemented appears" - [SQuirrel]

(a) Design Debt

(b) Requirement Debt

Figure 3: Wordcloud: Design debt and Requirement debt

Figure 4: Other type of debt

Design Debt	Requirement Debt
todo	implementation
hack	fixme
Needed?	xxx
workaround	configurable
Unused?	convention

Table 3: Design and Requirement debt frequent keywords

4.3 Algorithms

For this part, choosing an appropriate algorithm was the crucial part as it is the case of multiclass regression. First, we want to explore all possible algorithms for our model. Among them, the neural network and logistic regression were our first choice. Later, we generated different models with the following algorithms:

1. Neural Network
2. Logistic Regression
3. Ridge Classifier
4. Naive Bayes

5. Support Vector Classification (SVC) with Linear Kernel
6. Support Vector Classification (SVC) with Radial Basis Function
7. Perceptron (SGDClassifier with loss equal to perceptron)

5 Experimental Design

5.1 Experiments

To feed data into models, we first shuffled the data randomly and split train and test sets with 70% and 30% ratio. For all algorithms except neural networks we created a pipeline. The pipeline is used for assembling several steps which can be cross-validated together while tuning different parameters. Further, we transformed data in vector forms with TFIDF Transformer from sklearn. At last, we measure results with accuracy, F1 score, precision and recall values.

In the case of the neural network, we transformed comments data into vector formats and debt types into categorical value. Our network consists of an input layer and 1 hidden layer with rectified linear unit activation function, output layer with a softmax activation function. Here, we are only taking accuracy into the consideration because other 3 parameters (F1 score, precision and recall) were fluctuating in a large range and giving false positives.

Initially, we were implementing this project by writing scripting code. We weren't reusing code at much extent. Afterwards, for reusability purpose we decided to write clean code as much as we can and hence, we followed model-view-controller (MVC) framework and refactored into three categories: all cleaning and data operations in 'Code/DataOperations.py', machine learning models in 'Code/Model.py' and all visualization operations in 'Code/DataVisualization.py', leading to only one Jupyter notebook 'Main.ipynb' in the root project directory.

5.2 Evaluation

5.2.1 Which models perform better and pass our 75% benchmark?

Out of 7, only 3 (Neural Network, Ridge Classifier and SVC with the linear kernel) have just crossed the benchmark here. To achieve more accurate results, these models can be fine-tuned further.

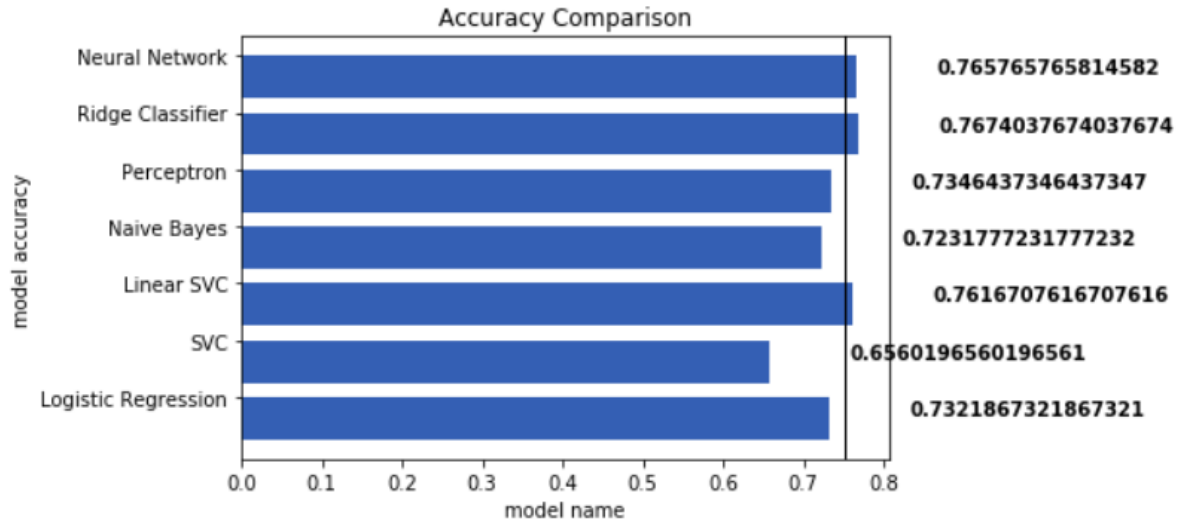

Figure 5: Comparison of models accuracy

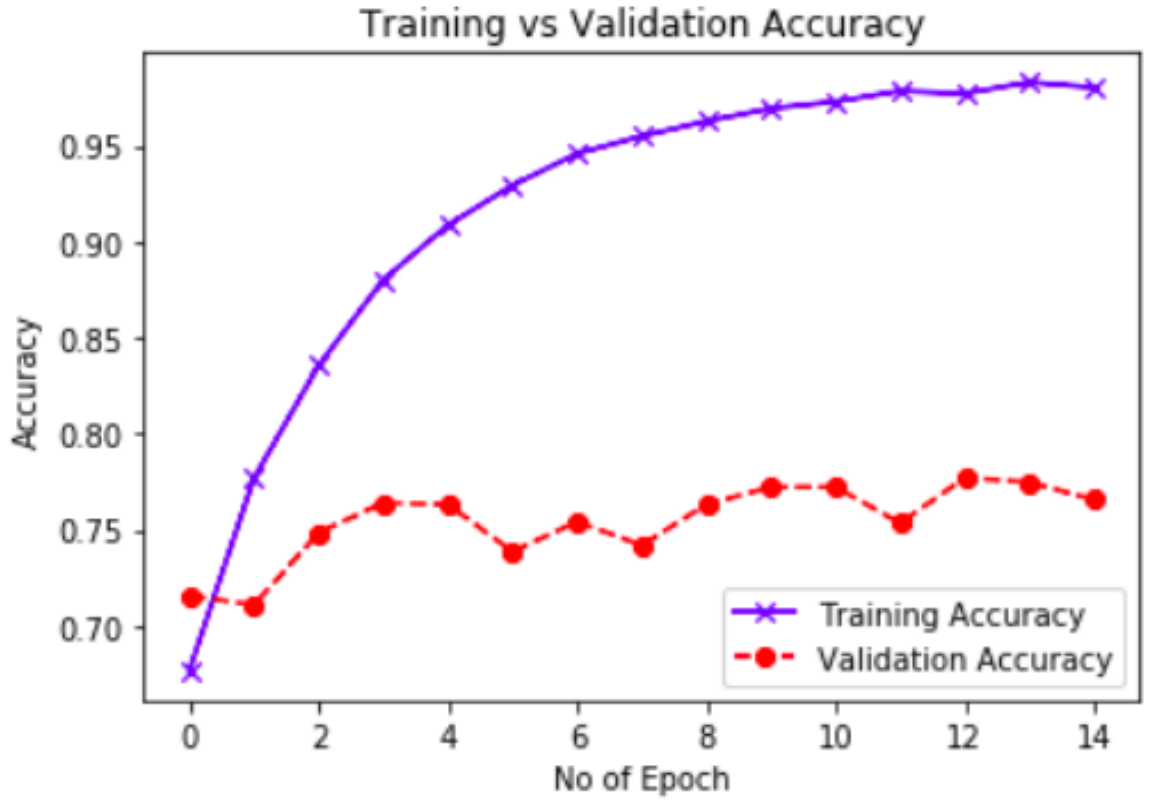

Figure 6: Neural Network: Train vs Validation curve

Model Name	Accuracy
Neural Network	**76.57**
Logistic Regression	73.21
Ridge Classifier	**76.74**
Naive Bayes	72.31
SVC with Linear Kernel	**76.16**
SVC with RBF Kernel	65.60
Perceptron	73.46

Table 4: Comparison of models accuracy

By comparing train vs validation results, we can say that overfitting is not applicable here. Neural Networks can be further utilized by adding more dense and complex layers in our case. One can also use pre-trained models (i.e VGG16 and so forth) on top another model and opt for various advanced learning methods (i.e. transfer learning, meta-learning) or complex models (i.e. convolutional neural networks, recurrent neural networks).

5.2.2 F1 score, Precision and Recall Values

Model Name	F1 Score	Precision	Recall
Logistic Regression	0.6623	0.7141	0.6320
Ridge Classifier	0.6719	0.7826	0.6192
Naive Bayes	0.4006	0.5388	0.4090
SVC with Linear Kernel	0.6938	0.7782	0.6459
SVC with RBF Kernel	0.2641	0.2187	0.333
Perceptron	0.6604	0.6784	0.6455

Table 5: Comparison of F1 score, Precision and Recall values

6 Conclusion and Future Work

In this course project, we are detecting two types of debt, design and requirement as they are the most significant types of SATD. The general procedure we followed in doing so is extracting source code comments from open source projects. These comments need pre-processing and filtering. The next step was manual classification in a dataset but instead, we used a dataset already available to us through the authors of the paper who conducted interviews and spent 185 hours just for the classification purpose. To generalize it, they use Cohen's Kappa coefficient which is one of the most popular constants to evaluate inter-rater agreement [1]. The level of agreement was found to be +0.81 which was significantly above the acceptance level i.e. +0.75. The next step was using a machine learning model to predict and detect technical debt.

Although results show promising F1 measures, one of the current threats to validity that we observed is that code may have some or no TD, but if it is not commented properly indicating the issue then it becomes quite difficult to detect any type of TD. So, a well-commented code is a necessity in order to detect TD. Moving further we tried some models such as logistic regression, linear svc, perceptron and also neural network. Even though there is some increase in accuracy, it is not too significant meaning we either need to allow time for some more fine-tuning and training or maybe deep learning won't have a significant effect. The last thing is that this project is the first step and needs to be further carried on for more research into machine learning models and to look into techniques for the removal of technical debt once it has been detected.

References

[1] E. S. Maldonado, E. Shihab, and N. Tsantalis, "Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt," vol. XX, no. X, pp. 1–20, 2017.

[2] E. Lim and A. Informatics, "A Balancing Act : What Software Practitioners Have to Say about Technical Debt," pp. 22–28.

[3] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an Ontology of Terms on Technical Debt," pp. 1–7, 2014.

[4] C. Seaman and Y. Guo, *Measuring and Monitoring Technical Debt*, 1st ed. Elsevier Inc., vol. 82. [Online]. Available: http://dx.doi.org/10.1016/B978-0-12-385512-1.00002-5

[5] N. Zazworka, R. O. Spinola, A. Vetro, F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," *International Conference on Evaluation and Assessment in Software Engineering*, pp. 42–47, 2013.

[6] N. S. R. Alves, T. S. Mendes, M. G. D. Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt : A systematic mapping study," vol. 70, pp. 100–121, 2016.

[7] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," vol. 56, no. 5, pp. 1–13, 2012.

[8] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the Impact of Design Debt on Software Quality," pp. 17–23, 2011.

[9] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the Impact of Self-admitted Technical Debt on Software Quality," 2016.

[10] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure It ? Manage It ? Ignore It ? Software Practitioners and Technical Debt," pp. 50–60, 2015.

[11] E. S. Maldonado and E. Shihab, "Detecting and Quantifying Different Types of Self-Admitted Technical Debt," pp. 9–15, 2015.

[12] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," 2007.

[13] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant : Identification and Removal of Type-Checking Bad Smells," pp. 329–331, 2008.

[14] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *J. Syst. Softw.*, vol. 84, no. 10, pp. 1757–1782, Oct. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2011.05.016

[15] N. Tsantalis, D. Mazinanian, and G. Panamoottil Krishnan, "Assessing the refactorability of software clones," *IEEE Transactions on Software Engineering*, vol. 41, pp. 1–1, 11 2015.

Appendix

File Name	Directory	Purpose
Main.ipynb	/	Main jupyter notebook
technical_debt_dataset.csv	/Dataset/ technical_debt_dataset.csv	Original Dataset
processedDataset.csv	/Dataset/ processedDataset.csv	Processed dataset by us
DataOperations.py	/Code/ DataOperations.py	For all filtering and cleaning operations
DataVisualization.py	/Code/ DataVisualization.py	For all visualization related operations
Model.py	/Code/Model.py	Contains all seven models

Table 6: Repo: File Details