POLITECNICO DI MILANO
Corso di Laurea Magistrale in Computer Science and Engineering
Scuola di Ingegneria Industriale e dell'Informazione

Self-Admitted Technical Debt Detection and Management in Issue Tracker Systems

A Machine Learning approach

Relatore: Prof. Mark James Carman

Tesi di Laurea di:
Cristian Giannetti
Matricola 920387

Anno Accademico 2021-2022

—*"Come va l'alternanza scuola-lavoro?"*
—*"Cosa?"*
—*"La vita"*

Maicol&Mirco

Sommario

Nello sviluppo software, la metafora del Debito Tecnico indica un compromesso atto a conseguire obiettivi di breve termine in un modo che può influire negativamente sulla salute e sulla manutenibilità di prodotti software sul lungo periodo. Il Debito Tecnico Self-Admitted Issue-based (SATD-I) è una sotto-categoria del Debito Tecnico che indica debito riconosciuto dagli sviluppatori e riportato in sistemi di tracciamento di issues. L'obiettivo di questo lavoro è di identificare il SATD-I (nello specifico debito di codice, debito di documentazione, e debito di test) dal testo, con sistemi di Natural Language Processing, e analizzare il comportamento degli sviluppatori a riguardo, studiando chi risolve le issues relative al debito tecnico, quante issues sono risolte, e quanto tempo è necessario per risolverle. Per rispondere a questi quesiti, sono stati utilizzati due modelli di Machine Learning, comparando tra loro i risultati ottenuti. Con 972 issues classificate a mano, abbiamo addestrato e validato una modello basato su Support Vector Machine, e uno basato su Logistic Regression, ottenendo un punteggio F1 di 0.678 per il primo, e di 0.7722 per il secondo. In seguito, abbiamo estratto un insieme di 1500 token dal modello di Logistic Regression, addestrato, a causa delle sue migliori performace, per mostrare come il SATD-I può essere identificato e spiegare i risultati del modello. Abbiamo poi classificato 2.3M di issues che coprono oltre 20 anni di sviluppo di progetti Apache e Mozilla, usando questi dati per capire qual è l'approccio degli sviluppatori verso il SATD-I, in confronto agli altri tipi di issues. I risultati hanno mostrato che la maggior parte dei prodotti hanno una percentuale di issues che identificano Debito Tecnico tra il 14.8% and 23.3%, che il debito di codice è spesso pagato nella prima settimana dalla sua documentazione, che la percentuale di debito pagato è tra il 68.86% e il 93.74%, e che gli utenti che creano queste issues sono anche i risolutori tra il 43.54% e il 93.08% dei casi.

Abstract

Technical Debt metaphor in software development introduces a compromise to deliver short-term goals in a way that can negatively affect the health and maintainability of products in the long term. Self-Admitted Technical Debt Issue-based (SATD-I) is a branch of Technical Debt that refers to acknowledged debt reported by software developers in issue tracker systems. The purpose of this work is to identify SATD-I (Code debt, Documentation debt, and Test debt) from text, using Natural Language Processing techniques, and analyze the behaviour of developers against it, by studying who resolves debt, how much debt is solved, how long is needed to resolve it. To do this, two different Machine Learning models were used and their results compared. Using 972 manually classified issues, we trained and validated an SVM and a Logistic Regression models, resulting on F1 scores of 0.678 for the former, and 0.7722 for the latter. Due to higher performance, we extracted a set of 1500 weighted tokens from the logistic regression trained model, that shows how SATD-I can be identified. We used this model to classify a dataset of 2.3M issues from more than 20 years of development of Mozilla and Apache projects, using these data to compare how developers approach SATD-I, against the rest of the issues. The results showed that most of the products have a percentage of Technical Debt issues between 14.8% and 23.3%, that Code Debt is often paid in the first week from its report, that the percentage of paid debt is between 68.86% and 93.74%, and that the creator of the issues also resolves it between 43.54% and 93.08% of the cases.

Contents

Chapter 1

Introduction

1.1 Overview

The study presented in this thesis aims to explore the field of automatic Technical Debt detection in software development using Natural Language Processing techniques, and its management by software developers. In particular, the focus of the research is the debt reported in issue tracker systems, creating an approach that is able to classify an issue as soon as it is created. To the best of our knowledge, this approach was explored once in 2022, and we aim to improve the reliability of the classification, to inspire some future work in this field.

The study was supervised by Prof. Mark James Carman of Politecnico di Milano[1].

1.2 Problem

The concept of Debt in software development was introduced to refer to low-quality deliverables that will likely need to be fixed (paid) in the future. We will see in Chapter 2.1 how Technical Debt is identifiable and how it affects the present and future work for developers.

As many studies have pointed out, if the introduction of Technical Debt seems to help the development in the short term allowing the software engineers to deliver their solution early, repaying the debt in the long run may be very expensive, and its cost increases with time. In fact, the sooner Technical Debt is repaid, the shorter is the time needed to actively repay it.

[1]Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Via Ponzio 34/5, 20133, Milano, Italy.

For example, talking about software Technical Debt, it is easy to notice how a quick and non-optimal solution may impact the future development: we know that the implementation of a fragile component will require revision in the future, but the more time that passes since the review, the more likely the component will be used in multiple parts. So, to modify it, it will be needed to check and test all the code blocks in which that component is used and to review all the design choices made around it.

From requirements collection to test writing, from design to release to production, every phase of software development may be affected by Technical Debt, and a bad decision in one of them affects all the work done after it.

The importance of identifying it at an early stage considers the possibility of saving resources by giving the chance of minimizing unnecessary code maintenance.

1.3 Goal

As we will see in Chapter 2, many studies explored the automatic detection of self-admitted Technical Debt - Code Based with several techniques. The main goal of the this research, instead, is to apply this field in issue tracker systems, studying an approach that is able to classify the software issues, to identify if they are related to Technical Debt or not.

Once we obtain a good classifier, it is possible to use its abilities to identify some trends about how Technical Debt is solved and who does it. Specifically, we will try to answer to the following questions:

- *RQ1: How can Technical Debt be identified from issues?* Creating a tool that identifies Technical Debt allows to classify more data, to have a better overview of the trends we are going to study.

- *RQ2: What type of Technical Debt is mainly paid in issues?* [Xavier et al., 2020] Among the existing types of Technical Debt, we can detect which is prioritised by developers.

- *RQ3: How do software developers resolve technical debt in issues?* To facilitate the search for an answer, this question is divided into three main sub-questions:

  - *RQ3.1 Who resolves Technical Debt tracked in issues?* Based on the data we collect, it is possible to check if the Technical Debt is resolved by those who reported the issue or by others.

  - *RQ3.2: How much Technical Debt tracked in issues is resolved?* Answering this question shows the attitude of developers towards Technical Debt, to understand how much importance they give to it.
  - *RQ3.3: How long does it take to resolve Technical Debt tracked in issues?* Tracking the time needed to resolve Technical Debt, it is possible to check how developers approach the issues they created compared to others.

The dataset used to answer these questions is 20-MAD [Claes and Mäntylä, 2020]. It contains over twenty years of information about 820 Apache and Mozilla open source projects. With 2.3M issues, it is a good representative of the way software developers work. Then, it will be possible, to compare some computed metrics with studies that were made considering different contexts.

1.4 Thesis Structure

After this introduction, the thesis is structured as follows:

- Chapter 2 presents the state of the art of the topic we are working on. It discusses which are the key studies and researches that prepare the base for this thesis.
- Chapter 3 describes in details how the study was designed and the shape of the data we use to perform the training, the validation, and the real world use case that will be used to extract the data to answer the research questions.
- Chapter 4 explains in details how the classifiers we consider for this study were designed and built, the metrics that we use to evaluate their performances, and a comparison between them.
- In Chapter 5 we discuss the results of the experiment and present the answers to the three research questions.
- Chapter 6 draws the conclusions of the research considering the aggregated results and presents some possible future work.

Chapter 2

State of the Art

2.1 Technical Debt

According to Avgeriou et al., "Technical Debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible" [Avgeriou et al., 2016].

The Technical Debt metaphor was firstly introduced by Ward Cunningham in 1992 in *The wycash portfolio management system* [Cunningham, 1992]. Cunningham put a developer's unfamiliarity with code and their obfuscated long-term view about the software they are working on as the primary reasons for creation of Technical Debt. As software evolves, it needs refactoring that takes into account features that didn't exist long ago, when the implemented solution seemed like the best possible one. In addition, if a developer is not fully familiar with the software and the technology they are working on, their implementations of new features may not be optimal given how little experience they may have. Again, as time goes by and people become more familiar with the application, the code will need to be reorganized to avoid losing functionality and to make sure the code is kept clean and easily maintainable, as if the developers exactly knew what they were doing all along.

Since its introduction, Technical Debt has been taken up in several studies and its meaning expanded to include contexts in which time savings are preferred at the expense of quality, like in presence of tight deadlines or different priorities.

Cunningham's observation considered the shipment of sub-optimal code like getting into debt. A little debt may help to speed up the development, but if it is not promptly repaid, the debt starts to grow soon. The meaning of "repaying" in this context is associated to the code rewriting to eliminate the sub-optimal solution in favor of a more stable and maintainable one.

Why does the debt grow over time?

When the debt lives long, the sub-optimal solution affects the future development of the software in a significant way. Due to inter-dependencies between components, each block of code will have an influence on future development choices. So, the more a weak solution survives, the harder will be to change it in favor of a better one. That's why we can consider the time passing as an interest on the debt.

According to Cunningham, unpaid debt can have a huge impact on entire organizations, and development approaches like the *waterfall model* aims to reduce debt problems by planning every single aspect of the software requirements. Unfortunately, these approaches in many cases are not optimal, for example when the software needs a continuous growth or when encountering a change in the requirements, and we will often end up dealing with some forms of Technical Debt.

Avgeriou et al. consider that the interest on debt is composed of two main parts: recurring interests and accruing interests [Avgeriou et al., 2016]. The first groups the indirect additional costs of the presence of Technical Debt, like the reduced velocity of development and the loss of quality that affects the maintainability of the software; the accruing interest, instead, considers the additional costs caused by not-quite-right code, that affects its evolvability.

Among the reasons of the introduction of Technical Debt in software, the early shipping is the most considered by the developers [Xavier et al., 2020]. Often, in fact, they prefer to quickly deliver what they are working on, due to deadlines or other time management issues. Several studies have, instead, found several other causes that involve all the phases of software development: starting from bad design decisions about structures, frameworks, technologies and languages [Verdecchia et al., 2020], unavailability of a key person and lack of information about a technical feature [Avgeriou et al., 2016], architecture violations [Martini et al., 2014], while, originally, Cunningham was referring to lacks of experience of the developers on a project

they were moved on and to a lack of flexibility of the software due to a not-too-long term view.

Types of Technical Debt

To better understand the origin of the Technical Debt and to optimise its prevention, several studies created a classification of Technical Debt based on its causes. One of the first works was performed by Tom et al., collecting information from multivocal literature reviews, from interviews to software developers and academics, and final reviews. The purpose of the study was to create a classification by dimension, ending up with the following division into five categories: Code, Design and architecture, Environmental, Knowledge distribution and documentation, Test [Tom et al., 2013].

Alves et al., then, broke and integrated this result into a set of more detailed categories, based on the nature of the debt as the main factor of classification, that allow to immediately identify which phase of software development the debt belongs to. The Technical Debt types defined in *Towards an Ontology of Terms on Technical Debt* [Alves et al., 2014] are the following [Magnoni et al., 2016]:

- *Architecture* debt: Technical Debt caused by architectural sub-optimal choices that compromised the scalability, maintainability or other aspects of the quality of the product.
- *Build* debt: Poor building process, with flaws in a software's build system that consume unnecessary time and resources.
- *Code* debt: Violated best practices and low quality code, like duplicated code, complex or inconsistent code, slow algorithms and low multi-threading quality.
- *Defect* debt: Refers to bugs and failures found but postponed due to tasks with higher priorities.
- *Design* debt: Poorly detailed design or violations of best practices about object-oriented design, like bad usage of patterns or imprecise structure of components to be developed.
- *Documentation* debt: Outdated, incomplete or lack of project documentation, such as code comments or user documentation.
- *Infrastructure* debt: Sub-optimal configuration of technologies, frameworks, inadequate programming languages and other development-related processes.

- *People* debt: People issues that can cause problems with the development activities. Examples are experienced people considered single point of failure due to a late or a lack of training of the team.

- *Process* debt: Refers to inefficient processes, like communication or development processes or others.

- *Requirements* debt: Gap between requirements of a software and its actual implementation, like non respected constraint or inconsistent assumptions.

- *Service* debt: Issues introduced by selection and substitution of web services.

- *Test automation* debt: Issues with the automation of the tests of old features to support continuous integration and faster development cycles.

- *Test* debt: Refers to shortcuts taken in automated acceptance, integration and unit testing. Examples are lack of tests, low quality tests, test errors or low coverage.

2.2 Self-Admitted Technical Debt

Self-Admitted Technical Debt (SATD) is a type of Technical Debt that is acknowledged and documented by developers. To report it, several methods are used, like comments in source code, or the usage of issue tracking systems, or any other other type of documentation that is written to support the software's life cycle. Even when these workarounds are detected by developers, a significant subset of them is never repaid, due to the *Rework Effort* that is needed that would lead to short term additional costs in software development [Mensah et al., 2016], even if the long term benefit may be higher.

Several tools exist to identify SATD from source code, but the most performing tool that we found is *SATD Detector* [Liu et al., 2018], built as an Eclipse plug-in to detect and track SATD comments from the IDE. The approach used to automatically classify the comments is based on Text Mining, and its performance was based on 212 thousands comments from 8 different open source projects, showing an F1-score between 0.518 and 0.841, with an average of 0.737 [Huang et al., 2018].

2.3 Issue-based Self-Admitted Technical Debt

Among Self-Admitted Technical Debt types, Issue-Based SATD (as known as SATD-I) has been gaining more and more interest in the recent times. Until 2022, SATD-I used to be manually identified. This highlighted evident limitations on the extraction of relevant examples to study, to understand how Technical Debt is treated by software developers. Recent studies that are focused on SATD-I compare the different types of debt found in issue tracker systems and the reasons behind its creation [Xavier et al., 2020], or try to understand when software developers identify Technical Debt and how they resolve it [Li et al., 2020], starting from a small set of issues to study, composed by a few hundreds of items.

In 2022, the publication of the research *Identifying self-admitted technical debt in issue tracking systems using machine learning* [Li et al., 2022] introduced the automatic detection of SATD-I, using CNN-based model that resulted in a F1-score of 0.686, studying 4,200 issues. To the best of our knowledge, up to now, this is the study that takes into consideration the biggest dataset used to detect SATD-I.

2.4 20-MAD Dataset

Answering the proposed research questions requires some real world data to understand how Technical Debt is treated by Software Developers. To reach this goal, we decide to use 20-MAD (20 Years of Issues and Commits of Mozilla and Apache Development) [Claes and Mäntylä, 2020] dataset, an open source dataset that contains detailed information about issues tracked in several Mozilla and Apache products. 20-MAD dataset includes information about 3.4M of commits, 2.3M of issues, and 17.3M of issue comments, all wrapped into 820 different projects, from 1994 to 2020. As we focus our research on issues, the data contained in this source matches perfectly our needs, as each issue is composed by the product they are part of, a summary, a description, anonymized data about the users who created and worked on it, the date of creation, resolution, and last update, and so on.

2.5 Hadoop/Camel Technical Debt Dataset

Examples of Technical Debt issues are reported in the dataset extracted for the study *Identification and Remediation of Self-Admitted Technical Debt in Issue Trackers* [Li et al., 2020], where 500 issues from Hadoop and Camel projects were extracted and classified for a similar purpose. The dataset

contains information about different types of Technical Debt, specifying for each issue the debt type (among the ones mentioned in 2.1) and sub-type.

2.6 Natural Language or Not (NLoN)

Natural Language or Not (NLoN) [Mäntylä et al., 2018] is an open-source R package for text analysis, that is able to recognize natural language. Mostly used in text pre-processing, it separates strings such as source code or stack traces from input text. The solution uses regular expressions, machine learning, and language detection techniques to achieve high performance in identifying natural language in different types of software engineering text inputs, including descriptions of issues.

2.7 TF-IDF

Term Frequency-Inverse Document Frequency (*TF-IDF*) is a numerical statistic used to calculate the importance of terms and keywords in a set of documents or a corpus, firstly introduced by Salton and Buckley [1988]. *TF-IDF* is the product of two values: Term Frequency identifies the frequency of the usage of a term in a document; Inverse Document Frequency, instead, represents the rarity of the same in the corpus. To calculate the *TF-IDF* score, the following formulas are applied:

$$TF = \frac{occurrencies\ of\ a\ term\ in\ the\ document}{\#terms\ in\ the\ document}$$

$$IDF = log(\frac{\#documents}{\#documents\ containing\ the\ term})$$

$$TF{-}IDF = TF * IDF$$

The obtained *TF-IDF* score is higher for terms that are rarely used across the corpus but often in the considered document, and lower for broadly used terms across all the corpus. This technique is commonly used in Natural Language Processing applications, like information retrieval or text classification, and can be adapted in classification of issue's titles and bodies.

2.8 Support Vector Machine

Support Vector Machine is a supervised machine learning model used for classification and regression analysis. Developed by Cortes and Vapnik [1995], it works by finding the hyperplane that best separates different classes of data in a feature space, being able to handle non-linearly separable data as well. Compared to non-linear SVMs, linear ones are more efficient and less prone to overfitting, and easier to interpret. There is no limit on the number of features that SVMs can handle, and they are considered robust to noise and outliers. In Natural Language Processing field, SVMs are used for information retrieval, sentiment analysis, and text classification, both binary and multi-class.

2.9 Logistic Regression

Logistic Regression is a binary classification model used in statistics and machine learning, that uses the probability of the class label starting from the input features, using a logistic function. The concept of logistic regression was firstly introduced by Cox [1970], as a way to provide the relationship between one or more predictor variables with a binary response. Compared to neural networks, Logistic Regression models do not have any hidden layer between input and output nodes, but a logistic activation function is applied to the output. However, they can be used as a part of neural network or deep learning architectures.

Chapter 3

Design of the study and input datasets

This chapter presents the input datasets we used to study and answer the research questions presented at the beginning of the thesis, and the design of the study. The main idea is to build a classifier, using Machine Learning, to help us identify the issues that describe Technical Debt. In order to do that, we used Natural Language Processing techniques to process two datasets of issues: one to train our classifier with (D1) and one that represents the real world dataset, from which we extract the statistics to explore how developers manage debt. Our classifier will be trained and evaluated with D1, and we must ensure that the data taken as input are optimised for our purpose. 20-MAD must be reviewed and cleaned as well, to let it be comparable with the training data.

The whole design and implementation are presented in details in the following sections.

3.1 Real World dataset

Issues reported in 20-MAD dataset are used in our study to finalize the results to answer the research questions. Data reported in the dataset are consistent and provide a perfect overview of software's life cycles, with historical and detailed information on how the work is organised by software developers. We exclude commits and comments from our study, and we just focus on issues, that have enough details for our research.

Every row of the *Issues.parquet* dataset in of 20-MAD represents an issue. A full list of attributes that we are interested in follows:

- *source*: it specifies if the issue comes from an Apache or a Mozilla product;
- *product*: name of the project for which the issue was created;
- *created*: timestamp of creation of the issue;
- *updated*: timestamp of last update of the issue;
- *last_resolved*: timestamp of last resolution of the issue (resolved issues can be reopened);
- *summary*: short text to identify the core of the issue (cannot be empty);
- *description*: text to describe the issue in details (can be empty);
- *status*: status of the issue (can be Open, Resolved, In progress, In review, Blocked, ...);
- *reporter_key*: identifier of the user who is the issue reporter;
- *creator_key*: identifier of the user who is the issue creator;
- *assignee_key*: identifier of the user who is the issue assignee;

Summary and Description are the two attributes that will be used to build the classifiers to detect Technical Debt, while the rest of the attributes will be used to perform studies about it. As some results will be grouped on product level, we decide to remove from the dataset the issues from the products that contain less than 500 items, to avoid highly biased results. By doing so, we discard 68,089 issues (2.3% of the total) from 477 products, leaving us with 2,246,038 issues grouped in 343 products. In the following sections and chapters, when we refer to 20-MAD dataset, we are considering its cleaned version that we just described.

3.2 Evaluation dataset and Debt types

Issues in 20-MAD dataset do not contain information about their category, if they refer to Technical Debt or not. For this reason, we must create the source that will be used as training dataset by ourselves. Starting from Hadoop/Camel Technical Debt dataset presented in section 2.5, we can adjust it to better fit our study. The dataset contains 383 issues that do not refer to Technical Debt, and 117 issues related to Technical Debt. Among the second group, the types of Technical Debt have very different frequency:

out of 117 Technical Debt issues, there are 59 that refer to Code debt, 33 to Documentation debt, and 28 to Test debt. All together, these three types represent 79% of the total debt found in the study. Architecture, Build, Defect, Design, and Requirement debts are represented 10 times or less.

We do not consider this dataset strong enough to support our training, and we prefer to expand it by taking random issues from 20-MAD dataset and classifying them manually, based on their summaries and descriptions. We decide to focus on specific types of debt because of two main reasons: the difficulty to find less frequent technical debt types, and the balance we want to keep among the categories in the training set, in order to reduce biases. The types we focus on are Code Debt, Test Debt, and Documentation Debt. From now on in this research, we consider the rest of the Technical Debt categories as *Non Technical Debt*, together with the issues that do not represent any type of Technical Debt.

We start by removing duplicates from Hadoop/Camel dataset (some issues refers to multiple Technical Debt types and are reported multiple times), remaining with 480 items, and we assign a binary value to each item (0 or 1) based on our definition of Technical Debt. Then, we extend this dataset to have around one third of Technical Debt issues and around one thousand elements in total. To reach this goal, we extract random issues from 20-MAD using python's *random*[1] library, until we reach 110 issues for each type of Technical Debt. By doing so, we complete our dataset with 972 issues: 330 Technical Debt issues and 642 Non Technical Debt ones. This dataset will be used to train and validate the models that we are presenting in the following pages.

3.3 Data pre-processing

Before implementing the classifier, the structure of the data must be defined. The goal of the first phase is to clean the data and remove what is not relevant for the classifier, from both the dataset we use: the one we trained our classifier with (training and test sets, i.e. D1) and the one representing the real world dataset we are going to classify (i.e. 20-MAD). Usually, in fact, raw data contains noise that should be managed prior to the classification, to avoid biased results.

The classification of the issues based on text requires special attention on the domain we are working on. The language used in the issues by the users is a technical language and it often contains blocks of code and logs or code-

[1]random: https://docs.python.org/3/library/random.html

specific words, like names of classes, methods, packages and frameworks. The challenge of the text pre-processing phase is to build a bias-free input for our classifier, or at least to reduce it at minimum.

3.3.1 Text cleaning

A classification based on domain-specific words (like the training set we are using) can reduce its accuracy when run on a much broader dataset. To reduce at minimum this risk, we decided to remove from our two datasets all the words and blocks that could badly interfere with the result of the classification. Moreover, since D1 is mainly composed by issues from *Hadoop* and *Camel* projects, we tried to filtered out all the words that were linked to these specific projects, as we considered them not to be relevant as input of the classifier.

To provide an example of each single step of the cleaning process we performed, we are going to show how a single item of the dataset would be processed during each step. The example issue we are going to process in the following pages is identified by these two fields:

Summary: *"[TEST] Remove hadoop.logfile.* and fix dependencies"*

Description: *"The package is only called in the root file and doesn't look like it is used anywhere at all. It causes the following message while building due to a conflict with TestPipelineUtils methods:*
[ERROR] PipelineUtils.java line 147 - getProfileUpdates failed with error code 1"

The resulting text after each step can be found in Table 3.1 for the Summary field and in Table 3.2 for the Description field of the example.

Removing code blocks and logs from text

Many issues report entire blocks of code or lines of logs, in some rare cases even thousands of lines. We considered the presence of these lines not to be relevant for the detection of Technical Debt with Natural Language Processing for several reasons. First of all, the blocks of code are not considered natural language, we cannot classify them with the same techniques we use for English text. If we ignore them, many strings or words related to specific programming languages or used to name variables would interfere with the input of the classifier. For example, the words *true* and *false* are present in many programming languages as Boolean values (both Java and Python use them, just to mention two programming languages); so, we cannot consider

their usage to be irrelevant compared to the usage of the words *true* and *false* outside of this context.

In the same way, logs are not reliable for the purpose of this research. They contain in general several lines of text, sometimes written as natural language, sometimes not. There is no difference for us between the lines composed by natural language from the others within logs because all the text in these cases is machine-generated and copy-pasted into the description of the issue. Several words are widely used in logs (like *error* or *warning*) and they would bias the data collection for the classifier.

The removal of code from the issues in the dataset D1 took place in two stages and it was focused on the description of the issues and not their title, as we assumed that the titles are not automatically generated or copy-pasted from code. We used an open source solution called NLoN (Natural Language or Not) [Mäntylä et al., 2018] to separate natural language from log messages or code. We processed each issue of D1 splitting its description by the new line character ($\backslash n$) and letting the NLoN predictor classify the obtained strings. The strings classified as Natural Language (*NL*) were then concatenated with a new line separator, while the others were removed, obtaining the new description of the issue.

After this computation, we manually reviewed all the issues in D1 to check if there still was any line of code that was not detected by NLoN, removing few lines of logs from 16 descriptions out of 972 issues.

The removal of code in 20-MAD was completely done using NLoN, with the same process described above. The manual check in 20-MAD was not performed due to the amount of data contained in the dataset, but we do not expect a relevant impact from the margin of error we have with NLoN. Tests made on NLoN showed good performance indexes for our purpose, achieving an area under ROC curve between 0.976 and 0.987 on three different data sources and between 0.913 and 0.98 in cross-source prediction. We can consider these performance indexes good enough for filtering the real world dataset we have. We explore in Chapter 4 why having extremely accurate data in D1 was much more important.

The description of the issue we took as example would be broken into two strings by the new line character and given as input to NLoN. The first string is considered natural language, while the second one is considered a log.

Cleaning symbols

The second phase of the cleaning process was performed to eliminate irrelevant symbols from single words. This section describes a process performed for both the title and the description of the issues. The cleaning started splitting the text (the whole title and the whole description) into tokens using any whitespace character as separator. Then, from each token we removed any symbol from its ends to avoid the influence of punctuation of any type and we removed any apostrophe in the token. The symbols were removed from the left and the right of the tokens because, while the punctuation marks are positioned in the right end of the words, characters like parenthesis can be found both on the left and on the right end of words. The removal was iterative because we can have ellipsis or multiple question marks or exclamation points.

In the presented example we see that the token *[TEST]* lost its square brackets, while *hadoop.logfile.** was reduced by two characters from right. In the description, the apostrophe in the token *doesn't* and all the punctuation marks were removed.

Removing irrelevant words

In the next phase the words that we considered irrelevant were filtered out from the dataset. We divided irrelevant words into extremely short words, stop words, and meaningless tokens.

We considered extremely short words as the ones composed by one or two characters. They were filtered out to avoid confusion between different abbreviations or acronyms, typos or other words that would interfere with the input of the classifier.

The stop words, instead, are words that are very common and do not add significant meaning to the text we are evaluating. Examples of stop words are *this*, *the*, *yours*, *while*, and they are generally ignored by search engines. To filter them out we used the *NLTK*[2] Python library to check, for each token, if it is considered part of the English stop words or not and, in case, we deleted them from the text. There is not a unique set of stop words but the one used by *NLTK* is one of the most used in Python and our research relies on it.

Meaningless words were represented by tokens with a structure that is not common in a natural language. We defined them as the tokens that contain one or more symbols enclosed by two letters (e.g. *alpha-beta*) or

[2]NLTK: https://www.nltk.org/

tokens that contain a number (e.g. *java8*). We decided not to split tokens containing symbols, and we preferred to delete them, because those tokens are likely related to code, like an attribute call (e.g. *array.size*) or a package (e.g. *import java.util.**). Our definition included some tokens with typos among the meaningless words (e.g. a missing space after using the comma) but unfortunately we are not able to recognize the intention of the authors, so we decided to discard these tokens as we assumed these cases to be rare enough not to affect the results of our study.

In the summary of the example issue, *hadoop.logfile* is considered a meaningless token due to the dot in the middle, while the word *and* is in the set of the English stop words. In the description, the words *is*, *in*, *it*, *at*, *to*, *a* are extremely short words as we defined them. The remaining ones that were deleted, instead, were stop words, and they are *The*, *only*, *the*, *and*, *all*, *while*, *with*.

Stemming words

The goal of the next phase was to transform the single words into their stem, so that the terms we have become independent of their usages. The purpose of this step is to reduce the used vocabulary into stem words so that we can compare and group words with same meaning used in different contexts with different inflections.

To do so, all the letters that compose the dataset were lowered. Then, the *NLTK* Python library was used to perform the stemming of the words. The algorithm used was PorterStemmer [Porter, 1997].

Removing little used words

The last step of the cleaning process was related to context-specific words. The purpose of this step is to avoid a language that is used in a very small subgroup of projects because it would distort the data used for the classification. The risk would be to give much weight to context-specific tokens, but we expect them to be words related to code or to names of services, not relevant for our goal.

This step was performed creating a dictionary with all the remaining stem words of D1 as keys and the number of different projects of 20-MAD in whose problems the words appeared as values. We decided to remove from D1 the tokens used in less than 80 projects out of 820. This step does not modify the content of 20-MAD, because the classifier we are creating is not influenced by little used words of the real world dataset.

From the example we are considering, the token *testpipelineutil* is context-specific. Probably, the reason is that it represents the name of a test class of a specific project, not used in many other projects of our dataset.

Phase	**Summary**
Initial text	*[TEST] Remove hadoop.logfile.* and fix dependencies*
Remove code blocks	*[TEST] Remove hadoop.logfile.* and fix dependencies*
Remove symbols	*TEST Remove hadoop.logfile and fix dependencies*
Remove irrelevant words	*TEST Remove fix dependencies*
Stemming	*test remov fix depend*
Remove little used words	*test remov fix depend*

Table 3.1: *Result of Summary field of example issue after each cleaning phase.*

3.3.2 Labelling Technical Debt

After the data cleaning, we decided to classify the issues of D1 assigning them a label based on the type of debt they represent. As specified above, we focused our study on three categories of Technical Debt, that are Code debt, Documentation debt, and Test debt. So, we added a binary label called *technical_ debt* to each issue to specify whether it is related to one of the three mentioned types of debt or not. A value equal to 0 was assigned to the issues that do not represent Technical Debt and the ones that describe Architecture debt, Build debt, Defect debt, Design debt or Requirement debt. For the others, we assigned a value equal to 1 to the *technical_ debt* field.

Phase	Description
Initial text	*The package is only called in the root file and doesn't look like it is used anywhere at all. It causes the following message while building due to a conflict with TestPipelineUtils methods:* *[ERROR] PipelineUtils.java line 147 - getProfileUpdates failed with error code 1*
Remove code blocks	*The package is only called in the root file and doesn't look like it is used anywhere at all. It causes the following message while building due to a conflict with TestPipelineUtils methods:*
Remove symbols	*The package is only called in the root file and doesnt look like it is used anywhere at all It causes the following message while building due to a conflict with TestPipelineUtils methods*
Remove irrelevant words	*package called root file doesnt look like used anywhere causes following message building due conflict TestPipelineUtils methods*
Stemming	*packag call root file doesnt look like use anywher caus follow messag build due conflict testpipelineutil method*
Remove little used words	*packag call root file doesnt look like use anywher caus follow messag build due conflict method*

Table 3.2: *Result of Description field of example issue after each cleaning phase.*

Chapter 4

Development of a binary classifier for text

In this chapter we present two Machine Learning models developed for our classification problem, trained and evaluated with the test and the validation sets we pre-processed in Chapter 3. The first model uses Support Vector Machine, while the second one is based on Logistic Regression. After the explanation of the implementations, we analyse the obtained results, to see where they perform better and which one we consider the more reliable for our research.

4.1 Evaluation of the models

To compare the performance of the models, we evaluate the classification of the validation data we described before, with each model, comparing the real binary labels of the issues with the ones assigned by the two models. A common approach used with binary classifiers is to presents the results with a confusion matrix, a table that shows numerically how many elements the model classifies correctly for each class. The matrix is composed by four numbers that indicate:

- True Negative (TN): number of issues not related to technical debt, correctly classified by the model;
- False Positive (FP): number of issues not related to technical debt, classified by the model as "Related to technical debt";
- False Negative (FN): number of issues related to technical debt, classified by the model as "Not related to technical debt"

- True Positive (TP): number of issues related to technical debt, correctly identified by the model.

These values, analysed singularly, may lead to inaccurate conclusions, especially when we deal with imbalanced dataset or when the consequences of a wrong classification are different between the classes. That's why we also use other indices that, based on the values of the confusion matrix, help us to better evaluate the two models. These indices are:

- Accuracy, indicates how often the classifier has a correct outcome.

$$ACC = \frac{TP + TN}{TP + TN + FP + FN}$$

- Precision, or Positive Predictive Value, indicates how often the classifier has a correct outcome when it assigns a Technical Debt label.

$$PPV = \frac{TP}{TP + FP}$$

- Recall, or True Positive Rate, indicates how often the classifier is correct when it deals with Technical Debt issues.

$$TPR = \frac{TP}{TP + FN}$$

- F1, that combines Precision and Recall to a single value, performing the harmonic mean between the two scores.

$$F1 = 2 * \frac{PPV * TPR}{PPV + TPR}$$

We use these rates to compare the results of the two models and see how they handle different classes of data, to see which algorithm would better perform to answer the research questions presented at the beginning of the document.

4.2 Classification with Support Vector Machine

This section describes the implementation of the model based on Support Vector Machine (SVM), used for the classification. To use this technology, new operations need to be performed on the training dataset to optimise it as input of the classifier. Next, the performed actions on the data and the implementation of the classifier are described, and some considerations on the first results are presented.

	abil	abl	access	account	adjust	...
0	0	0	0.056	0	0.21	...
1	0	0	0.31	0.18	0	...
2	0	0.3	0	0.42	0	...
3	0.29	0	0	0	0	...
...	...	...	...	...	...	...

Table 4.1: *Example of a tf-idf matrix. The rows represent the indexes of the documents, the columns represent the terms contained in at least one document.*

4.2.1 Text to numeric values

To evaluate text with a Support Vector Machine, we need to find a meaningful way to represent words as a numeric values. A broadly used approach in Natural Language Processing field is to use tf-idf (Term Frequency - Inverse Document Frequency) to compute which are the terms that characterise each class.

Assigning numeric values to terms

Applying the tf-idf to the training dataset, we compute two sparse matrices composed by terms (t) as columns, and issues (d) as rows, one for summaries and one for descriptions. The matrices are then filled with the tf-idf scores ($tfidf(t, d, D)$, being D the set of documents, i.e. the issues in our case). If the term t_i is not contained in the document d_j, then $tfidf(t_i, d_j, D)$ score is equal to zero. An example of the structure of the obtained matrix is visible in Table 4.1.
For each *(t, d)* couple, we now have a value that represents the importance of the term t in the issue d, compared to all the other issues.

Our next objective is to separate the terms that most likely identify technical debt from the ones that identify other issues. This is achieved by summarising into a single score for each term the statistics we just computed. For each term of the table, the score is calculated by the average of all its tf-idf scores, taken positive if the related document is labelled as technical debt, or negative if it is not. Due to the imbalance of the training set, each tf-idf value was divided by the percentage of the class it was part of (0.346 for Technical Debt issues, 0.654 for Non Technical Debt issues). We now have a score for each term, negative or positive, that represents how

likely it is to be correlated with debt. The highest the score, the more likely it is.

Assigning numeric values to issues

To shape the input of the SVM, we decided to assign two numeric values to each issue: one representing its summary, and one its description. The two values are calculated as the sum of the scores we just computed for each term contained in the summary and the sum of the ones contained in the description, separated. Terms in the validation set that were not met in the training set have a score equal to zero. The main consideration to take into account to optimise these values is related to the length of the issues and the uniqueness of the used terms. Summaries are in general shorter than descriptions, and rarely contain the same term twice, while descriptions can be very long and detaild, or even empty strings. In the algorithm we built to compute these values, repeated terms were considered only once because their tf-idf scores already represent their volume of usage in the text. The length of the texts, instead, is not considered to be a problem. Considering that we have both positive and negative scores for terms, long texts have a broader range of values compared to short ones. This should not cause problems to the algorithm we are using, as a very high or very low scores are caused by a high usage of terms related to a specific classes of issues, and their prediction should then be expectedly straight forward.

4.2.2 Building the classifier

After we shaped our training data, we can create the binary classifier that predicts the Technical Debt label of the issues we want to classify. The input we obtain is composed by triplets of values: a decimal number for the summary of the issue, another one for its description, and a binary value label that represents the class to which the issue belongs to. As we mentioned above, the expectation we have with the way we shaped the text is that Technical Debt issues are represented in general with higher scores, while the others have lower scores, both for summaries and descriptions. So, distributing them in a plane, we should see the the two clusters occupying different parts of it. The projection of the training data on a bi-dimensional plane is visible in Figure 4.1.

As we can see from the figure, the model we built seem to well divide the two classes on the training data, with the Technical Debt issues that mostly occupy the Quadrant I of the plane, while the others mostly occupy the Quadrant III. We can also notice a high number of dots on the X-axis,

Figure 4.1: *Representation on a plane of the training dataset*

with a description value equal to zero: most of them represent issues that do not have a description, but only a summary.

The model we chose for the implementation of the binary classifier is a Support-vector machine because of its performance with classification, as we are trying to divide the two classes of issues on a bi-dimensional space. Due to the structure of the input data, a linear classification adapts well to our problem and is able to optimise the parameters of the hyperplane used as margin to give the right importance to summaries and descriptions. In fact, we need some computation in order to understand what is the role that single features have on the classification.

4.2.3 Preliminary results

The validation of the model with the test data allows us to verify the performances of the classifier we built. To do that, we assign numeric values to summaries and descriptions of the issues that compose the test set, as we did for the training set, using the weighted tf-idf scores computed earlier. In Figure 4.2 we can see how the test data is distributed in the summary/description scores plane and how the decision regions are divided between the two classes of issues we have. Table 4.2 represents, instead, the confusion matrix for the data of the graph, containing the number of issues for each class and how they are classified.

From the confusion matrix we obtained, we can compute the performance

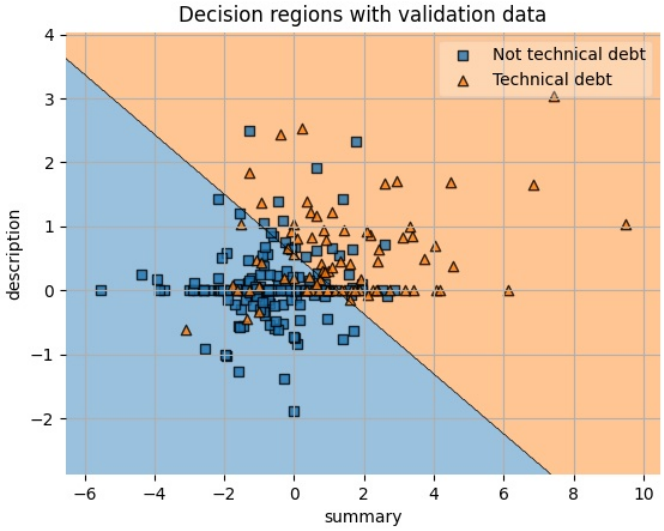

Figure 4.2: *Decision regions with test data*

		Predicted label		
		Negative	Positive	Total
Real label	Negative	133	29	162
	Positive	25	57	82
	Total	158	86	244

Table 4.2: *Confusion matrix for test classification with SVM.*

indexes described in section 4.1 to better understand the strengths and weaknesses of this model. The number of TP recognised by the SVM model is 57, and it represents almost the two thirds of the issues predicted as Technical Debt related. The value of Precision is in fact 0.6628. The number of FN is lower than FP, and so, value of Recall is higher than Precision and is equal to 0.6951. It means that almost 70% of issues labelled as Positive are correctly identified and labelled by this model. The obtained $F1$ score is equal to 0.6786.

In Table 4.6 we can see how the Technical Debt issues of our validation set are classified, grouped by Debt type and indicator. Comparing the number of TP with FN, we can notice how much the model is able to classify the different types of debt. The performance is good on Code debt, where the algorithm correctly recognised 79% of the issues, while on Documentation debt the correct classifications are 20 out of 28, that is 71%. With Test debt, instead, the model does not perform well. Only 56% of the issues

Debt type	Debt indicator	FN	TP
Code	Complex code	0	1
Code	Dead code	1	7
Code	Duplicated code	0	4
Code	Low-quality code	3	6
Code	Multi-thread correctness	0	3
Code	Slow algorithm	2	2
Documentation	Lack of documentation	0	3
Documentation	Low-quality documentation	4	9
Documentation	Outdated documentation	4	8
Test	Lack of tests	7	8
Test	Low coverage	1	0
Test	Low-quality test	3	6

Table 4.3: *Number of FN and TP predicted by SVM model divided by Technical Debt type.*

are classified as Technical Debt, 14 out of 25. Among the Test debts, a reasonable classification is performed on Low-quality tests, but the Lack of tests and the Low coverage are not easily predicted by our SVM model.

SVM linear model is not based on randomness. So, we can run the same algorithm using only the summaries of the issues as input, to evaluate how much the single features contribute to the final classification. We do not perform the same experiment using only the descriptions because it is considered an optional field, that is not present in many of the issues of the dataset, and it would require a balance of training and test dataset that would not allow us to have a fair comparison of the final results. Running the same algorithm providing only summaries as input, the classification of the validation set ends with a Recall value equal to 0.6585 and a Precision equal to 0.6667. $F1$ score is 0.6626. These values do not differ much from the original indexes. Compared to the previous results, the Precision score is very similar, but the Recall is lower by 0.0366 points, due to a lower number of TP, and a higher number of FN as a consequence. When present, the value added by the descriptions is low but significant. The reason of such a high importance of summaries may be their structure, because they should only contain meaningful words to represent the core of the issues, to let developers identify immediately what is the main goal and topic with few words. Descriptions, instead, are more broad and more difficult to classify

just giving a score to the words that are used in the text, as the algorithm just described does.

4.3 Classification with Logistic Regression

The model we use to compare our previous results with is a logistic regression model that we build. Also in this case, we need to reshape the data to fit the input of the new model. In the next sections we describe how the training and validation data were processed and what shape and parameters were used for the model, before the evaluation of the results.

4.3.1 Input of the model

The input chosen for the input nodes are the tf-idf scores for summaries and descriptions of the issues contained in the training set. The terms considered for the input are the top 750 for each feature, ordered by term frequency, chosen to let the model have a number of nodes in the input layer equal to 1500. So, some terms selected to evaluate the descriptions are different from the ones for the summaries, because we expect them having different structures.

Other options were evaluated to shape the input of the model in a different way, but then discarded. The first one was to use the Bag-Of-Words vectors, a list of binary values that represent the presence of terms in the text. We considered it to be less effective than tf-idf because of the loss of information about the text we are evaluating. With Bag-Of-Words, in fact, we do not know how many times a term is present inside a text, which can be relevant for the evaluation. The second option we discarded was the merge of summary and description into a single text feature, so that the input of the model could be a tf-idf representation of a single text. Due to the different structures of summaries and descriptions, we considered this option not to be powerful due to the generalisation of the usage of terms. If a term is present in the text of a summary, in fact, can be probably considered much more important than the presence of the same term in the text of a description. The reason we believe this to be possible is the role of summary, that is written to encapsulate the main point of the issue, and often it is then shorter then descriptions. Having them separate lets the model decide if the inputs have different relevance or not.

4.3.2 Building the classifier

The model used for the classification is a single layer MISO (Multiple Input-Single Output) network with 1500 input and a single output. No hidden layer was added due to the structure of the output, that is linearly separable for the final evaluation, and hidden layers are usually used for non-linear separations. The activation function used is a Sigmoid, that fits the binary classification we perform with the logistic regression. The described model gives us the probability for a certain element to belong to each class, so that we can use that number to make a prediction to label that issue. The label assigned to each element is the class the element has the highest probability to belongs to.

To setup the model and run the experiment, PyTorch[1] library was used.

4.3.3 Preliminary results

The number of epochs our classifier runs is chosen to minimise the loss on the validation dataset. In Fig 4.3 we can see the graph of the Validation Loss function, that reaches a flat trend after about 400 epochs. In few trials we made to tune the parameters of the network, the trend starts to slightly rise after about 500 epochs, but stopping the run before the absolute minimum of the loss function lets us prevent to overfit the model on the training data.

At the end of the training, the model is ready to classify the test data to evaluate its performance. The confusion matrix in Table 4.4 shows the results of the validation from which we can extract some performance indices. Almost three quarters of the Technical Debt issues are classified as positive, having a Recall equal to 0.7439. Out of 76 positively-predicted elements, only 15 are FN, and the Precision index is 0.8026. So, $F1$ score is equal to 0.7722, significantly higher than the score obtained with SVM.

		Predicted label		
		Negative	Positive	Total
Real label	Negative	147	15	162
	Positive	21	61	82
	Total	168	76	244

Table 4.4: *Confusion matrix for test classification with Logistic Regression.*

[1]PyTorch: https://pytorch.org/

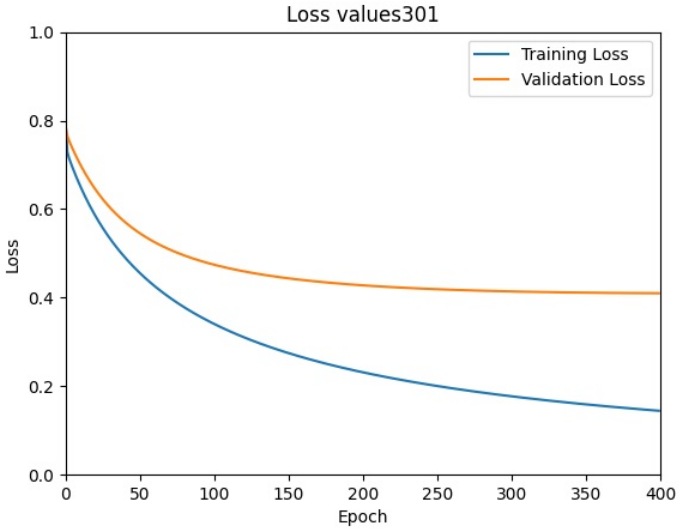

Figure 4.3: *Representation on a plane of the training and validation loss functions.*

From Table 4.5, we can see the details of the classification. Code Debt is the hardest debt to be detected by the developed logistic regression classifier: 66% of the issues in this category is correctly classified. Among subcategories, we have bad performance on *Complex code*, *Duplicated code* and *Slow algorithm*, while *Multi-thread correctness* from our validation set is fully detected. Moving on, Documentation Debt has a better result, with 79% of the issues classified as positive, with very good results on *Outdated documentation* and *Lack of documentation*. The highest score is obtained by Test Debt, composed by 80% of true positive on the validation data. *Low coverage* debt was not detected, but it is validated on only one element, so its percentage is not considered meaningful for the evaluation of the results.

As we already mentioned above, each input node of the classifier handles the tf-idf score of a single term, whether it is in the summary or in the description of the issues. At the end of the training phase, for each input we can extract the weight tuned by the model to explore which are the terms that most influence the outcome of the prediction. We should not consider those terms as the most significant ones, as their weights are also influenced by their tf-idf scores in the dataset, i.e. by their frequencies. The highest weights among the 750 tokens related to summaries, that most likely identify technical debt, are the following: *test*: 6.45; *document*: 5.44; *doc*: 5.20; *improv*: 3.91; *cleanup*: 3.46. Other meaningful terms among the

Debt type	Debt indicator	FN	TP
Code	Complex code	1	0
Code	Dead code	2	6
Code	Duplicated code	2	2
Code	Low-quality code	3	6
Code	Multi-thread correctness	0	3
Code	Slow algorithm	2	2
Documentation	Lack of documentation	0	3
Documentation	Low-quality documentation	4	9
Documentation	Outdated documentation	2	10
Test	Lack of tests	3	12
Test	Low coverage	1	0
Test	Low-quality test	1	8

Table 4.5: *Number of False Negative and True Positive Technical Debt issues predicted by Logistic Regression model divided by Technical Debt type.*

top 10 are *inconsist*, that is intuitively linked to technical debt, and *javadoc*, another term that identifies documentation. The reason of so many terms related to documentation may be the fact that, if present in the summary, these terms likely identify the core of the issue. So, when an issue related to documentation is open, it is likely not because of software failures, or new feature requests, but it may be related to a needed improvement or a missed update of the software documentation. Among the description tokens, some of the met top scores have a similar impact on the detection of Technical Debt. For example, the weights assigned to *doc* and *document* are 3.32 and 3.27 when present in the issue's description. Due to the fact that the descriptions are generally broad and more descriptive compared to summaries, similar weights end to have a different impact on the categorization of issues, as their tf-idf scores may vary based on the whole description. Among the negative weights, the lowest ones in the summaries are *receiv*: -3.59; *fail*: -3.06; *empti*: -2.998; *languag*: -2.94; *work*: -2.69; the token *receive* appears only twice in the summaries of the training set, both related to errors on received messages, and both non technical debt related. The token *fail* is usually related to software failures, expected not to be technical debt, and the token *empti* often represents unexpected values returned by functions: again software failures.

4.4 Comparison of the models

	SVM	**LR**
True positive	57	61
True negative	133	147
False positive	29	15
False negative	25	21

Table 4.6: *Confusion matrix for Support Vector Machine model (SVM) and Logistic Regression model (LR).*

Debt type	**Debt indicator**	**SVM**	**LR**
Code	Complex code	100%	0%
Code	Dead code	88%	75%
Code	Duplicated code	100%	50%
Code	Low-quality code	67%	67%
Code	Multi-thread correctness	100%	100%
Code	Slow algorithm	50%	50%
Documentation	Lack of documentation	100%	100%
Documentation	Low-quality documentation	69%	69%
Documentation	Outdated documentation	67%	83%
Test	Lack of tests	53%	80%
Test	Low coverage	0%	0%
Test	Low-quality test	67%	89%
Total		**70%**	**74%**

Table 4.7: *Percentage of True Positive issues, among all positive, predicted by Support Vector Machine model and Logistic Regression model, divided by Technical Debt type.*

Comparing the results of the two models, we notice that the Logistic Regression model have higher performance overall, as each value of its confusion matrix is better than the SVM approach's one, with a higher F1 score, equal to 0.7722 against 0.6786 of the SVM model. Focusing on positive items, the Recall values give us a performance index on Technical Debt issues, and it is visible from our data that the Logistic Regression model classifies correctly almost 5% more of them. Taking into consideration the single Debt categories, instead, we can see discordant results in some of them. The SVM model performs better on Code debt (79% of correctly classified debt issues

against 66%), a bit worse on Documentation debt (71% versus 79%) and significantly worse on Test debt (56% for the SVM model and 80% for the Logistic Regression model).

Among the 5 most frequent types of debt, the Logistic Regression model has significantly better performance on Lack of tests, Outdated documentation and Low-quality test, while the results for Low-quality code and Low-quality documentation are equal for the two models. Immediately after, ranking the items by frequency, we find Dead code subcategory, better identified by the SVM model for 1 issue; then, the next two subcategories have again the same results with both the models. So, if we consider the frequency of the types of debt, we find again that our Logistic Regression model performs better than the SVM model.

Chapter 5

Experimental Evaluation

Looking at the data presented in Chapter 4, we identify the Logistic Regression as a more reliable model, with better performance on several aspects of the classification. To answer the research questions of this study, we use the results of the classification of 20-MAD dataset using the Logistic Regression classifier we built, that we find to be the most accurate among the described ones.

5.1 RQ1: How can Technical Debt be identified from issues?

We demonstrated how text can be used to identify Technical Debt. Our Logistic Regression model let us classify issues using 1500 pre-identified tokens. These tokens are divided into two groups, Summary tokens and Description tokens, that were used as input labels of the nodes of the binary classifier. Every token has a weight assigned by the model, that was tuned during its training, and these weights can let us understand which are the ones that can most likely label an issue. This set of tokens can be further extended using a bigger dataset to train the model.

Some of the tokens in the two groups are the same, as they were found in both description and summary during the classification, but they have been assigned different weights, due to the difference in structure of summaries and descriptions. While the first one should identify the core of the issue in a few words, the latter can be broad and full of details to describe the problem, and it can mention the related tasks, the needed steps to resolve an issue, and so on. A word does not have the same importance in the identification of Technical Debt if used in the summary or used in the description of an issue.

Table 5.1 and Table 5.2 contain the top ten tokens with highest and lowest weights for Summary and Description. In the two sets we can detect some similarities, like *doc* and *document*, both present among the top tokens, or *clean* and *cleanup*, the first one among the Description's tokens, the second one among the Summary's ones. Other tokens, instead, have very different weights between the two sets, like *extend*, (3.8069 among summaries and -0.8507 among descriptions), or *improv* (3.9118 among summaries and -0.74189 among descriptions).

Summary TD tokens	**Weight**	**Description TD tokens**	**Weight**
test	6.4485	root	3.9995
document	5.4352	clean	3.5577
doc	5.1959	config	3.4452
improv	3.9118	doc	3.3175
mani	3.8773	document	3.2721
extend	3.8069	print	3.1746
cleanup	3.4584	size	2.9688
javadoc	3.4431	row	2.9246
inconsist	3.2619	thread	2.9190
rewrit	3.1673	dynam	2.6284

Table 5.1: *Top ten tokens that identify Technical Debt issues with relative score, from Summary and Description.*

In figure 5.1 it is possible to see the distribution of the weights for the two groups of tokens. The computed weights for summaries have a broader range of values, from -3.5853 to 6.4485, with an average score of 0.0346, while the ones for the descriptions vary from -3.4368 to 3.9995, with an average score of 0.1196. Overall, the two distributions follow the same trend.

Summary Non-TD tokens	Weight	Description Non-TD tokens	Weight
receiv	-3.5853	stack	-3.4368
fail	-3.0584	mapreduc	-3.0102
empti	-2.998	click	-2.7957
languag	-2.9362	jvm	-2.6944
work	-2.6859	stuff	-2.5716
custom	-2.6636	log	-2.5645
http	-2.5932	box	-2.5068
profil	-2.5419	run	-2.4959
search	-2.5251	bodi	-2.2827
version	-2.4991	environ	-2.1827

Table 5.2: *Top ten tokens that identify Non-Technical Debt issues with relative score, from Summary and Description.*

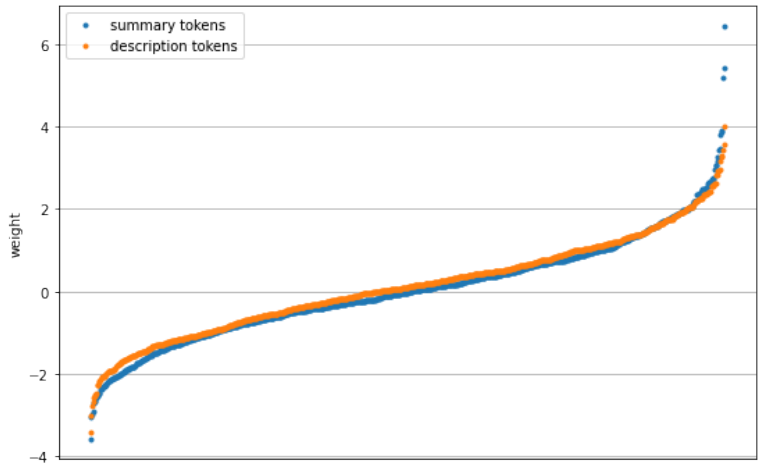

Figure 5.1: *Weight distribution of Summary and Description tokens, sorted. Every dot represents the weight of a single token.*

Our binary classifier can be used to identify Technical Debt from the real world dataset we pre-processed, and perform some studies on the output. By feeding the classifier with the td-idf score of the identified tokens for summary and description of every single item, one by one, we can classify all the issues. The result of the labelling for the issues from 20-MAD is summarised in Figure 5.2. The graph shows the distribution of the rates of Technical Debt issues for each product. The products taken into account are

the ones that have at least 500 issues reported in the dataset, to reduce at minimum the noise in the results. We can see that half of the projects have a percentage of issues related to Technical Debt between 14.8% and 23.3%, with a median equal to 18.9%. The project with the minimum percentage is Plugins Graveyard with 3.56% of Technical Debt issues, while the one with the maximum value turned out to be Mozilla QA with 37.14% of issues that refer to Technical Debt.

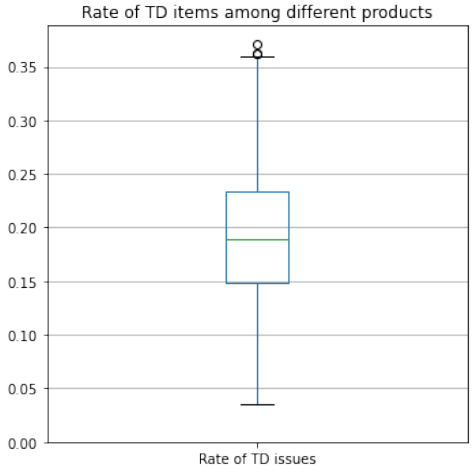

Figure 5.2: *Rate of Technical Debt issues grouped by product.*

5.2 RQ2: What type of Technical Debt is mainly paid in issues?

The types of Technical Debt we can compare in this section are the ones taken into account for our study: Code Debt, Documentation Debt and Test Debt. To understand how these categories are treated by software developers we can target the training dataset of our study, where we can get the issues divided by debt type. For each category, 110 issues were considered, and they are presented in Table 5.3. We divide the Technical Debt issues by debt type. The issue statuses are divided into *Paid* and *Not paid*: the first encloses issues with status "Done", "Resolved", or "Verified", while the second includes all the others. Counters and percentages are shown in the table too.

Comparing the results, we can notice that *Documentation* Debt is the most paid category compared to the others, with 39 issues resolved out of 110, equal to 43.6%. Between the remaining two types, *Test* Debt has the

Debt type	Issue status	Counter	Percentage
Code	Paid	39	35.5%
Code	Not paid	71	64.5%
Documentation	Paid	48	43.6%
Documentation	Not Paid	62	56.4%
Test	Paid	44	40.0%
Test	Not Paid	66	60.0%

Table 5.3: *Counter and percentage of Paid and Not paid Technical Debt, divided by category.*

highest score, with 40.0% of resolved issues, while *Code* debt is resolved for the 35.5%, having probably less priority compared to the rest.

From the mentioned paid debt, we extracted data about the time needed to resolve the issues. Table 5.4 describes how much time was needed to resolve the issues, dividing the data into four time sections that are: less than a week; between a week and a month; between a month and a year; more than one year.

Days	Code	Documentation	Test
0-6	42.9%	37.1%	37.5%
7-29	17.9%	34.3%	25.0%
30-365	25.0%	5.7%	25.0%
>365	14.2%	22.9%	12.5%

Table 5.4: *Time needed to resolve Technical Debt, by type.*

We can notice that in the first week of detection, the highest priority is given to Code debt, that is resolved within 6 days for almost the 43%, while the other categories are equally at 37%. In the first month, Documentation debt is the most paid one, as more than 70% of the issues is resolved between 0 and 29 days, while Code and Test debt are slightly above 60%. The last two, in the same way, are treated equally in the first year after reporting the issue, both with 25% of issues resolved between one month and one year. 22.9% of the Documentation debt is paid after more than one year, meaning that if not solved in the first month, these types of issues are hardly resolved in 365 days.

5.3 RQ3: How do software developers resolve technical debt in issues?

To identify how Technical Debt is resolved, we used the real world dataset classified by the previously described Logistic Regression model, to compare how software developers treat Technical Debt and Non-Technical Debt issues. First, we check how much Technical Debt is resolved for each product (considering only products with highly reported issues, to reduce noise), relying on the binary classification performed by the model. Then, we compare who usually resolves the Technical Debt. As last, we compare the time needed to resolve Technical Debt issues, that gives us an idea on how much paying debt is prioritised among teams.

5.3.1 RQ3.1: How much Technical Debt tracked in issues is resolved?

The first sub question makes us compare the quantity of issues resolved. Tracking how much Technical Debt is solved gives us an idea on the importance that is given to its repayment. Table 5.5 summarise the number and percentage of issues identified as related to Technical Debt by our model, for ten different products from the real-world dataset we classified. The table contains a counter of the Technical Debt Issues for each product, the number and percentage of resolved Technical Debt issues, and the percentage of TD issues that were not resolved. Issues identified as *Solved* are the ones which have a status equal to "Done", or "Resolved", or "Verified".

Product	**# Issues**	**# Solved**	**% Solved**	**% Remaining**
SOLR	3,213	2,322	72.27%	27.73%
MESOS	3,362	2,315	68.86%	31.14%
HADOOP	3,543	2,940	82.98%	17.02%
AMBARI	3,817	3,506	91.85%	8.15%
FLEX	3,837	3,394	88.45%	11.55%
HDFS	4,212	3,359	79.75%	20.25%
FLINK	4,450	3,654	82.11%	17.89%
HIVE	5,093	3,737	73.38%	26.62%
HBASE	6,373	5,455	85.60%	14.40%
SPARK	9,356	8,864	94.74%	5.26%

Table 5.5: *How much Technical Debt is resolved, divided by product.*

Among the considered products, *SPARK* is the one with the highest

percentage of repaid Technical Debt, with the 94.74% of issues in a Complete state. On the other side, *MESOS* has the lowest percentage of resolved issues, that is 68.86% for the ones that represent Technical Debt. In all the products, most of the Technical Debt is actually repaid, and four of the mentioned products have a percentage of open issues that is less than 15% (AMBARI: 8.15%; FLEX: 11.55%; HBASE: 14.40%). This is an indicator that software developers are conscious of the importance of repaying Technical Debt, and that the related issues are most likely resolved when identified.

5.3.2 RQ3.2: Who resolves Technical Debt tracked in issues?

Our real world dataset allows us to verify who works on and repays Technical Debt. In the dataset, anonymized information about creators of the issues, reporters, and solvers, are present. We extracted these data from the ones we identified as related to Technical Debt, reporting our findings in Table 5.6. The table presents data about 10 different products that in the original study were extracted from Jira, as not all the products have these information available in 20-MAD. In the table we find the number of Technical Debt issues that were marked as resolved, and the number of them that were assigned to creators of the issues, or to their reporters, or to other people. As can be noticed, sum and percentage of the issues in each row do not match the total count of resolved, because a user can act both as creator and reporter of the same issue. The *Others* column counts the issues that are not assigned to creators nor to reporters of the issues.

The presented data shows very similar values for issues assigned to creators and to reporters for all the mentioned products. This can be partially explained by the issue-tracker system used to extract these data. Jira, in fact, during the creation of a new issue, automatically sets as *Reporter* the user who is creating it. This field can be then modified manually, if needed, but it requires an active action from the creator, and we can assume that the *Reporters* column is biased. Percentages of Technical Debt issues resolved by Creators or Reporters, instead, have very different values among the products. *SPARK* is the one with the lowest percentage of issues assigned to Creators (43.54%) or to Reporters (43.72%), while the percentage of issues assigned to thirds parties is 52.44%, one of the highest values among the selected products. On the other side, *FLEX* product has the highest percentage of Technical Debt issues assigned to *Creators* and *Reporters* (93.08% and 93.02%), and 6.92% to *Others*. From these data, we can also compute the percentage of Technical Debt issues assigned to Creators not Reporters and to Reporters not Creators, that are minimal due to the premises we

Product	# Solved	Assigned to					
		Creators		Reporters		Others	
		#	%	#	%	#	%
SOLR	2,322	1,062	45.74%	1,059	45.61%	1,260	54.26%
MESOS	2,315	1,066	46.05%	1,076	46.48%	1,237	53.43%
HADOOP	2,940	1,759	59.83%	1,775	60.37%	1,163	39.56%
AMBARI	3,506	3,204	91.39%	3,170	90.42%	302	8.61%
FLEX	3,394	3,159	93.08%	3,157	93.02%	235	6.92%
HDFS	3,359	2,380	70.85%	2,376	70.74%	975	29.03%
FLINK	3,654	1,991	54.49%	1,992	54.52%	1,657	45.35%
HIVE	3,737	2,869	76.77%	2,852	76.32%	864	23.12%
HBASE	5,455	3,673	67.33%	3,666	67.20%	1,782	32.67%
SPARK	8,864	3,859	43.54%	3,875	43.72%	4,648	52.44%

Table 5.6: *Who resolves Technical Debt issues, divided by product.*

did about the origin of the data. They can be computer with the following formula:

$$\%Creators_{NotReporters} = 1 - \%Others - \%Reporters$$
$$\%Reporters_{NotCreators} = 1 - \%Others - \%Creators$$

Of those 43.54% of Creators, in *SPARK* 3.85% of the Technical Debt issues are assigned to Creator that are not also Reporters, and 4.03% are assigned to Reporters and not Creators. For *FLEX*, instead, only 0.06% of Technical Debt issues are assigned to Creators Not Reporters, but none is assigned to Reporters that are not also Creators of the issues. Generally, we can see that three products have slightly less than half of the Technical Debt issues assigned to Creators and Reporters (SOLR: 46%; MESOS: 46%; SPARK: 44%), and seven that have more than half assigned to them (HADOOP: 60%; AMBARI: 91%-90%; FLEX: 93%; HDFS: 71%; FLINK: 54%-55%; HIVE: 77%-76%; HBASE: 67%). In general, from our data, we cannot identify a pattern that shows who are the developers who work on self-admitted Technical Debt. Based on product, creators of the issues and their reporters can be the assigners of most of these tasks (up to 93.08% of issues), while the percentage of third users that have these issues assigned fluctuates in a high range, between 6.92% and 54.26%.

5.3.3 RQ3.3: How long does it take to resolve Technical Debt tracked in issues?

Time needed to resolve Technical Debt is another indicator of the priority that is given to it. To better understand the data, we compare the results with the same extracted from Non Technical Debt issues, to study how software developers behave when dealing with each type. Figure 5.3 shows the distribution of the average resolution time, grouped by products, for Technical Debt and Non Technical Debt issues.

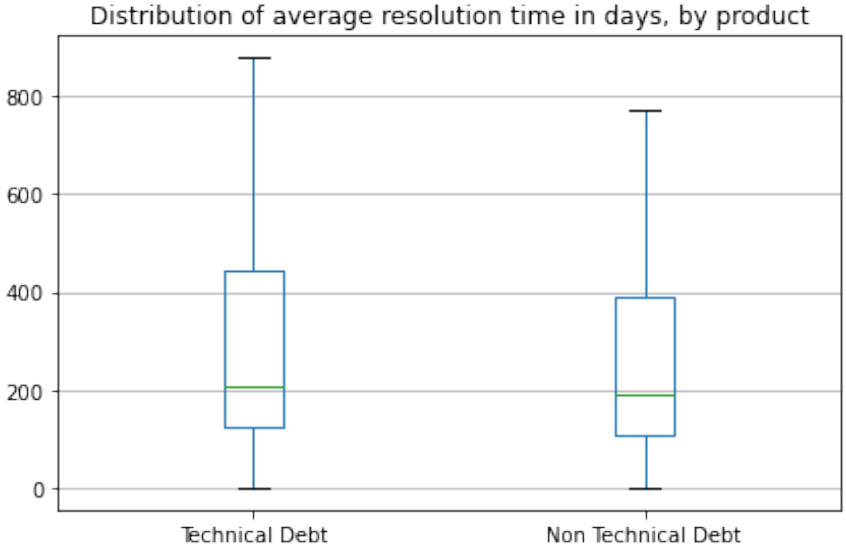

Figure 5.3: *Distribution of average resolution time in days for Technical Debt and Non-Technical Debt issues, grouped by product.*

For Non Technical Debt issues, the boxplot that is shown is slightly more compressed, indicating that all the values are distributed in a smaller range. 50% of the products have an average resolution time for Technical Debt issues between 126 and 445 days, with a median of 207 days, while for Non Technical Debt issues, developers of 50% of those products needed between 111 and 389 days on average to complete them, with a median equal to 193 days. An overview of these data shows that Non Technical Debt issues take more priority overall compared to Technical Debt, but we can see more detailed data in the following graphs: Figure 5.4 and Figure 5.5 represent the distributions of the rate of issues resolved in less than a week, between a week and a month, between a month and a year, and more than a year, for each product, divided into Technical Debt (5.4) and Non Technical Debt (5.5) issues.

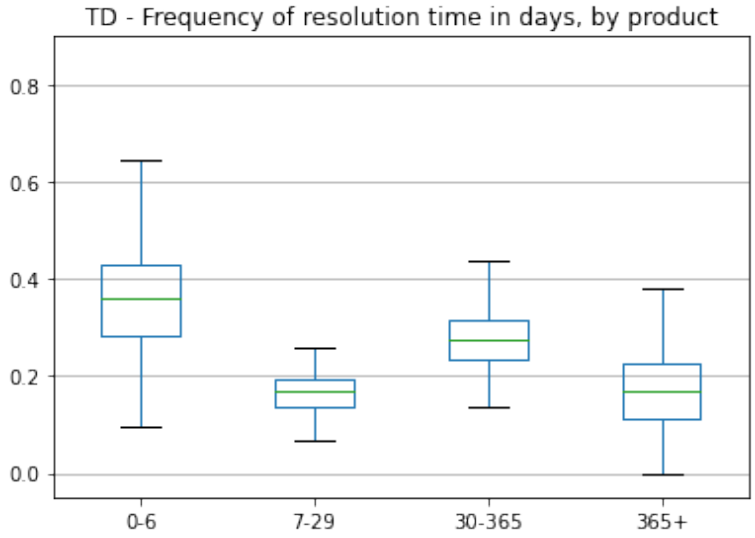

Figure 5.4: *Frequency of resolution time for Technical Debt Debt issues, in days, grouped by product.*

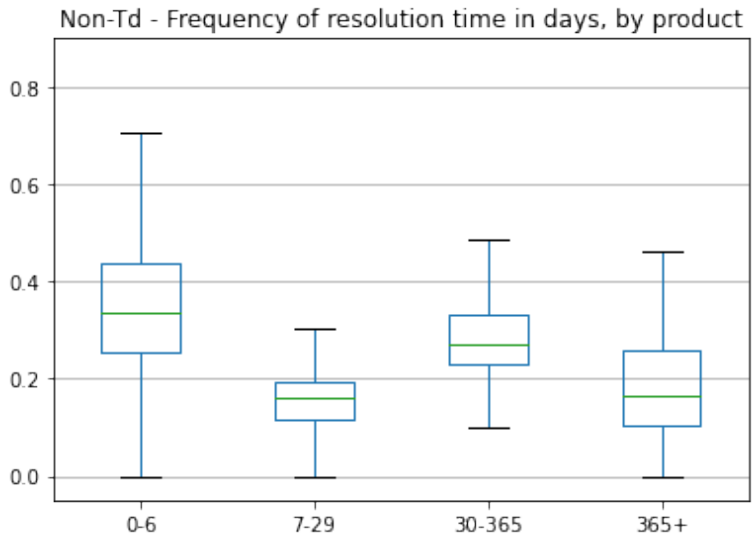

Figure 5.5: *Frequency of resolution time for Non-Technical Debt Debt issues, in days, grouped by product.*

From the two graphs we can notice the difference between the time needed to resolve the two types of issues. In half of the products, the Technical Debt issues resolved in less than week are between 28.2% and 42.9%, with a median equal to 36.2%, while the rest of the issues are resolved between 25.3% and

43.6%, with a median 33.7%, in half of the products, showing that the first are preferred to be the ones solved in a very short time. The second box plot in each figure represents issues resolved between a week and a month. Technical Debt issues have the first quartile equal to 13.8%, a median equal to 16.9%, and the third quartile is 19.2%. On the other side, Non Technical Debt issues have first quartile, median and third quartile equal to 11.6%, 16.2%, and 19.5%. Again, even in medium-short time, the priority seems to be given to Technical Debt issues. Moving on, issues resolved between 30 and 365 days have the mentioned values equal to 23.2%, 27.4%, and 31.5% for Technical Debt, and 23.0%, 27.2%, and 33.3% for Non Technical Debt. In this case, the box plots are very similar, with a slightly lower third quartile in the first case. Finally, Technical Debt issues resolved in more than one year have first and third quartiles equal to 11.2% and 22.6%, while for Non Technical Debt are 10.23% and 26.0%, and the medians are 16.7% for the first one, and 16.4% for the latter. From the data we extracted, Technical Debt issues seem to slightly take priority over the rest in the first month of their report into the ticket tracker systems, but they need more time on average to be resolved. Even if there are more issues that are resolved in less than 30 days, the ones resolved in more than one year take much longer compared to Non Technical Debt issues in the same category.

Chapter 6

Conclusions

In this work, we created a study about Self-Admitted Technical Debt - Issue Based, to understand how it can be identified and how software developers approach it. We now present our original contribution and the future research that can be done to support and further improve the Technical Debt management in software life cycle.

6.1 Original Contribution

The analysis of the management of Technical Debt requires an important amount of data from different sources to reduce at minimum the bias of the study and ensure generalizable results. With this purpose, we labelled the biggest dataset of issues as Technical Debt related or not, classifying 2.3M of issues from 820 different Apache and Mozilla projects, building the biggest dataset of issues containing Technical Debt information. To do this, we built and compared the results of two different classifiers, using Support Vector Machine and Logistic Regression, to have a good approach to classify these issues. Of these two classifiers, the first one, with an F1 score of 0.6786, was able to almost match the performance of a recent classifier with the same goal, built and published by Li et al. [2022], with an F1 score of 0.686. The second model, instead, has considerably higher performance, as it reached an F1 score equal to 0.7722. From this model, we extracted a list of tokens with a weight attached that helps to explain the model, showing which are the words that most likely identify Technical Debt issues or Non-Technical Debt issues. After the labelling, we showed that most of the products have a percentage of Technical Debt issues between 14.8% and 23.3%, with a median equal to 18.9%, while outliers can go up to 37.14%. In terms of priority, among the types Code, Documentation, and Test, the second one

is the most likely to be paid, but the first one is the quickest one to be resolved, as almost 43% of the resolved Code debt is paid in the first week of its reporting. Depending on the products, the percentage of paid debt varies between 68.86% and 94.74%, showing that often its resolution is considered a priority by developers. In terms of who resolves these issues, the results fluctuate in a wider range. Comparing the different projects, we saw that the creators and the reporters of the debt issues can also be the ones who resolve it between 43.54% and 93.08% of cases. Taking time into consideration, we also noticed that Technical Debt issues take more time to be resolved on average compared to Non-Technical Debt ones. On the other hand, higher percentages of debt issues are resolved in the first week and in the first month of their report, showing that, when they are resolved in more than one year, debt issues take much more time compared to the rest.

6.2 Future Work

The main limitations of the described work are related to the size of the dataset used to train the model and their performance review. Unfortunately, the process of manual labelling of the single issues does not allow to increase the scale of the training dataset. To confirm the results obtained by this study, the two models described in Chapter 4 can be trained and validated against bigger dataset, like for example the ones labelled by software developers in private company's projects. Often, these datasets already contain issues with a Technical Debt label attached, as several teams explicitly dedicate part of their work-time to repay debt, to avoid to find themselves working on an unmaintainable software in the long term. Unfortunately, the access to similar dataset for research purposes was not granted to us. Moreover, the training can be expanded using n-grams instead of single tokens to identify the context in which some words are used, that would increase the reliability of the models.

The whole study on Technical Debt performed as part of this research, in the future should also include all the types of Technical Debt identified by Alves et al. [2014], as our study was focused on Code, Documentation, and Test debt. Even if these often represent most of the debt present in issues, results can significantly change when considering other types of debt, like Infrastructure Debt or Architecture Debt, that can often require more time to be addressed compared to simple unit tests or a code change.

This research provides insights to Self-Admitted Technical Debt - Issue Based management and how developers approach it. As we saw, debt repayment occupies a significant part of software developer's work, and ad-hoc

solutions to identify and keep track of it, and measure its impact, can be one of the next challenges that issue tracker systems can have. Knowing the impact of an implemented workaround after weeks, months, or years, before its repayment, can give awareness on the importance of Technical Debt mitigation and save time and money on software development.

Bibliography

Alves, N. S., Ribeiro, L. F., Caires, V., Mendes, T. S., and Spínola, R. O. (2014). Towards an ontology of terms on technical debt. In *2014 Sixth International Workshop on Managing Technical Debt*, pages 1–7.

Avgeriou, P., Kruchten, P., Ozkaya, I., and Seaman, C. (2016). Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports*, 6(4):110–138.

Claes, M. and Mäntylä, M. V. (2020). 20-mad: 20 years of issues and commits of mozilla and apache development. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 503–507, New York, NY, USA. Association for Computing Machinery.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Mach. Learn.*, 20(3):273–297.

Cox, D. R. (1970). Support-vector networks. volume 297, pages 273–297.

Cunningham, W. (1992). The wycash portfolio management system. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, page 29–30, New York, NY, USA. Association for Computing Machinery.

Huang, Q., Shihab, E., Xia, X., Lo, D., and Li, S. (2018). Identifying self-admitted technical debt in open source projects using text mining. *Empirical Softw. Engg.*, 23(1):418–451.

Li, Y., Soliman, M., and Avgeriou, P. (2020). Identification and remediation of self-admitted technical debt in issue trackers. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 495–503.

Li, Y., Soliman, M., and Avgeriou, P. (2022). Identifying self-admitted technical debt in issue tracking systems using machine learning. *Empirical Software Engineering*, 27.

Liu, Z., Huang, Q., Xia, X., Shihab, E., Lo, D., and Li, S. (2018). Satd detector: a text-mining-based self-admitted technical debt detection tool. pages 9–12.

Magnoni, S., Di Nitto, E., and Tamburri, D. A. (2016). An approach to measure community smells in software development communities.

Mäntylä, M. V., Calefato, F., and Claes, M. (2018). Natural language or not (nlon): A package for software engineering text analysis pipeline. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 387–391, New York, NY, USA. Association for Computing Machinery.

Martini, A., Bosch, J., and Chaudron, M. (2014). Architecture technical debt: Understanding causes and a qualitative model. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 85–92.

Mensah, S., Keung, J. W., Bosu, M. F., and Bennin, K. E. (2016). Rework effort estimation of self-admitted technical debt. In *QuASoQ/TDA@APSEC*.

Porter, M. F. (1997). *An Algorithm for Suffix Stripping*, page 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Salton, G. and Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. *Information Processing Management*, 24(5):513–523.

Tom, E., Aurum, A., and Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516.

Verdecchia, R., Kruchten, P., and Lago, P. (2020). Architectural technical debt: A grounded theory. In Jansen, A., Malavolta, I., Muccini, H., Ozkaya, I., and Zimmermann, O., editors, *Software Architecture*, pages 202–219, Cham. Springer International Publishing.

Xavier, L., Ferreira, F., Brito, R., and Valente, M. T. (2020). Beyond the code: Mining self-admitted technical debt in issue tracker systems. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 137–146, New York, NY, USA. Association for Computing Machinery.

Acknowledgments

Vorrei ringraziare chi ha permesso, in un modo o nell'altro, che questa tesi giungesse al termine, dopo un lungo travaglio che ha dato alla luce una serenità ed un senso di liberazione mai visti prima.

Grazie a Maria Pia, per la semplicità con cui mi ricorda che questa vita vale la pena di essere vissuta.

A Stefano, caro amico e saggio punto di riferimento, senza il quale questo percorso non sarebbe probabilmente mai iniziato.

A Michela, sincera amica, che è riuscita a far stravolgere dei mediocri punti di vista radicati da decenni.

Al Prof. Carman, che ha avuto fiducia nel progetto, e che ha dato un supporto fondamentale alla conclusione di questo lavoro.

A Papà, Mamma, Nicolino, Elisabetta, Milena, e al loro affetto incondizionato, pronti a dare tutto senza chiedere mai. E allo stesso modo ad Alessandra, Alessandro, Lorenzo.

A Giorgia, Jacopo, Mattia, Niccolò, Riccardo, che mi ricordano l'importanza della curiosità, e che ad ogni parola o sguardo mi fanno capire sempre più l'unicità di ognuno di noi.

A Davide, Federico, Francesco Riccardo, Jacopo Pio, Stefano, che hanno accompagnato questo percorso universitario, in modi molto diversi, e che sono entrati con decisione nella vita di tutti i giorni, consolidando un'amicizia che non mostra pieghe nemmeno a migliaia di chilometri di distanza.

Ad Alessandro, Alice, Annaluce, Cinzia, Giacomo, Luiza, Matteo, Michele, Sabato, per aver dato e per dare ancora oggi colore ad una città grigia che attrae e prosciuga talenti.

A Celeste, Danilo, Enrico, Francesco, Isadora, Manuela, Silvia, e a Don Giorgio, perché da quindici anni e più, sono ancora oggi un esempio per i valori e le passioni che sono riusciti e riescono ancora a trasmettermi.

E ad amici, colleghi, e conoscenti, che in questi anni si sono interessati allo stato di questo lavoro con pensieri, suggerimenti, domande. Ogni singola parola ha avvicinato questo progetto alla sua conclusione.