Analyzing the evolution of Technical Debt together with DevOps metrics

A quantitative case study using Natural Language Processing

Daniel Skryseth

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2022

Analyzing the evolution of Technical Debt together with DevOps metrics

A quantitative case study using Natural Language Processing

Daniel Skryseth

Abstract

Background: Technical Debt (TD) is a metaphor for sub-optimal solutions that are assumed to affect the velocity of software development. It is therefore important to understand how it is possible to estimate TD accurately and then track it, as well as look at how it evolves and is prioritized.

Objective: This thesis aims to quantify TD issues from the sentiment of developer discussions, which will then be compared with DevOps metrics to observe how TD correlates with velocity. It will also be looked into how this approach may make it possible to track technical debt issues over time, such as how TD issues are prioritized and ultimately resolved.

Method: A quantitative case study has been conducted for this thesis. Data from Github and Jira were collected from five different open source projects. Issues from the data are classified based on a sentiment analysis using machine learning for Natural Language Processing (NLP), which are then related to their corresponding code and followed over time. The DevOps metrics are also measured from the same data and used together with the quantified TD in a correlation analysis.

Results: The results show that it is possible to use NLP to classify TD issues from developer discussions. Issues may then also be made relational so that they can be connected to their corresponding code and discussions across Jira and Github. The classifier in this thesis outperformed other similar studies and estimated that TD issues accounted for %10.3 - %17.06 of the issues in the projects. Comparison of open TD issues with velocity DevOps metrics over time showed meaningful correlations between deployment frequency and open TD issues, as well as some mixed correlations between lead time for changes and open TD issues. TD management also appeared to occur in short-term bursts of high frequencies.

Conclusion: As other studies suggest, quantifying TD issues using NLP to classify developer discussions can help identify TD issues that may otherwise be difficult to detect with static code analysis. It may also be useful for keeping track of TD issues in large projects to see how they evolve, be used for TD prioritization, as well as related to relevant code. However, one important drawback is that the TD issues are only detected when developers start discussing them. Lastly, the findings from the correlation analysis seemed to indicate that open TD issues can have implications for the velocity of software development if they are not actively managed.

Acknowledgements

Writing this master's thesis has been a challenging and incredibly rewarding process. Not only has it made me grow professionally, but also as a person. I could not have been without the excellent guidance of my supervisor Antonio Martini, for whom I am immensely grateful. His continuous support, compassion, patience, and extraordinary expertise are something that has truly impressed me.

I would also like to extend a special thank you to Doctoral Research Fellow Karthik Shivashankar, who has taken the time to help me with guidance. I must also thank fellow master's student Arsalan Khalid who took the time to help me verify sample data for the thesis.

Finally, I must express my profound gratitude to my girlfriend Elisabeth for all her support, patience, encouragement, and love.

Daniel Skryseth
May, 2022

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software quality reflects the quality of software based on how it is designed to meet structural and functional requirements [12]. Producing software is often a continuous effort, as software developers usually aim to maintain their software and facilitate the continuous delivery of new software features [46]. Developers who help meet these requirements will typically use tools such as issue tracking systems and version control systems to support the various processes involved.

These tools are then commonly used to systematically coordinate the priorities for the software, which makes them convenient to use for discussions related to the production and maintenance of software. Previous studies have found that one of the things discussed within such tools is Technical Debt (TD) [7, 9, 37]. This addresses software quality directly and serves as a metaphor for sub-optimal solutions that may benefit developers in the short term, but at the expense of long-term quality [23].

The causes of TD are underlying problems that are introduced either deliberately or inadvertently through simple and quick solutions, instead of choosing optimal approaches [50]. For example, ad-hoc architectural design, code quality concerns, or other poor design choices. If TD is not fixed by reworking the software, it is expected to affect the velocity of the software development. This includes making it harder or even impossible to add new features, as well as more difficult to fix the TD in the future [4].

Attempts have been made to measure TD for different scenarios, to examine how it can be prioritized and managed, and to look at possible consequences that may arise from it [6]. However, previous studies have shown that research on TD is lacking in certain areas, for example in prioritizing it [26, 35]. The latter is something that, in terms of needs including fixing bugs and developing new features, addresses the concern about whether and when TD should be addressed in a software project.

Software practitioners may use other specific tools to help their efforts improve the quality of their software, and thus make decisions related to TD prioritization. One of the most widely used tools for prioritizing TD has been SonarQube, which has been adopted by more than 100,000 organizations [1]. The way a tool like SonarQube works is that it inspects code quality and security through a statistic analysis, doing so by checking whether the code complies with a set of known rules.

However, this does not take into account the overall effect that TD may have, since tools such as SonarQube can only provide a preliminary overview of the TD through static analysis. There is therefore a greater need to understand TD further, which includes how it will be possible to more accurately estimate and follow TD issues over time [28]. In this way, it will be possible to look at how one can relate to the TD in a project, what effects it has, how it is prioritized, and the resolution of it.

Some studies have looked into additional methods that can be used to estimate TD more precisely. Lenarduzzi et al. [28] attempted to use a data-driven approach for estimating TD interest, comparing its relationship with the lead time from resolving Jira issues. The results from this indicated that there is a further need to explore such an approach, by seeing how data-driven approaches can be used for TD prioritization. Additionally, this may then also be used as an approach to see how TD correlates with other aspects of software development such as the velocity.

In another study by Ozkaya et. al [37], Natural Language Processing (NLP) was used with Machine Learning (ML) to detect TD issues from developer discussions. In their study, they stated that the approach could potentially be useful for either understanding or defining TD, such as identifying TD issues that could otherwise be hard to uncover with static code analysis alone. However, they went on to conclude that there was a need to further refine the method, which includes increasing its accuracy so that it could be used to more accurately quantify TD using NLP.

This thesis aims to explore these limitations further. In this case, by further exploring the connection TD has to developer discussions inside issue tracking systems and version control systems. These discussions may then be quantified as TD issues using a sentiment analysis with ML for NLP. Additionally, the issues may also be made relational across the different systems so that the TD issues can be tracked over time, which may lead to insight into how TD evolves from developer discussions, as well as provide an understanding of how it is prioritized and fixed.

[1]https://www.sonarqube.org/about/

After the TD issues have been quantified, they can be analyzed together with the performance of software development. This thesis will specifically look at how TD correlates with the velocity of software development, which has been measured by DevOps metrics [13, 15]. Both the quantified TD and DevOps metrics will be able to capture the evolutionary processes of a software project. Studying them together may then reveal insight into how TD can affect the velocity of software development over time.

The research questions (RQs) for this thesis are defined as the following:

RQ1: How can natural language processing be applied to developer discussions to quantify the evolution of TD?

Rationale: This RQ will try to improve the method from Ozkaya et. al [37]. Particularly by further exploring how natural language processing can be used to quantify TD from discussions, so that this new measure may be used with data-driven approaches for TD prioritizations.

RQ2: How can the evolution of TD discussions be analyzed by being correlated with DevOps metrics to provide insight into projects?

Rationale: By using the quantified TD as a data-driven approach with DevOps metrics, it may give insights into how TD correlates with the performance of software development, such as the velocity.

Chapter 2

Background

This chapter presents both the relevant theoretical context and previous findings, which aims to further explain the objective of this thesis. Firstly, an introduction to software development life cycles will be given, as this will affect how software practitioners relate to software development. Secondly, open source software will be briefly explained, since this is where data from this thesis will come from. Lastly, an introduction to both technical debt and natural language processing is presented, as these will be relevant for the research context of this thesis.

2.1 Software development life cycle

A Software Development Life Cycle (SDLC) is a process used for the production and maintenance of high-quality software. Breaking down a software project into different parts will limit and define descriptive work phases for a project. This could, for example, include something like a deployment- or evaluation phase. The final goal of an SDLC is to produce, maintain and modify software that satisfies customer expectations.

While an SDLC can be unique and adapted for a specific project. It is most commonly split into phases that can be seen in a whole range of frequently used models [43]. Each one of these models may be completely different from the next but, based on the type of project it is used for, capable of guiding software practitioners and organizations with their projects. The phases found in any such models usually cover a phase for some sort of planning and requirements gathering. Then, one phase for designing, software development, and testing respectively. Finally, one for the deployment, operations, and maintenance.

Although an SDLC, along with its different phases, may not summarize a project perfectly. It may serve as a conceptual framework, which can help visualize how it is possible to tackle problems, allocate workload and resources, as well as prioritize and manage the life cycle of a software project. There are a plethora of such SDLC models, where each may describe different project management methodologies. However, common models include such as the waterfall, spiral, unified process, extreme programming, v, incrementing, and agile models [44].

In recent years, agile methodologies, such as Scrum and Kanban, have become some of the most popular to use [24]. In agile methodologies, even though they may differ from each other, the development of a project is feedback-driven and executed in stages (often called iterations). This makes it possible to deliver software with continuous improvement, which may sustain a sort of evolutionary development. Making it so that project contributions can be continuously added in different iterations. For example, by adding new or updating previous features, fixing bugs, and so forth.

This is different from the traditional ones such as the waterfall model, which will instead have a sequential flow that is completely linear, where each phase is separated from the other and carried out one after another. For example, by separating and differentiating between the development and testing so that one phase may only be started after the first one has been fully completed.

2.2 Open source software

For this thesis, an SDLC will be a reference to open-source software development (OSSD). OSSD is the process of both developing and maintaining projects that are open-source [47] which may, based on their licenses, also be referred to as free/open source software (F/OSS). These are projects that are often based on some sort of agile methodology, and that has made either part or all of their entire source code freely available.

The latter is usually done by publishing source code and other content that are related to the F/OSS, so that it can be used, modified, or redistributed. This makes it so that contributors to OSS projects can be part of an SDLC with an evolutionary development life cycle. Particularly for this thesis, it will be looked into OSS projects that have made both their issue tracking system and code repositories available. These two concepts are further explained in the methodology of this thesis.

2.3 Technical debt

As briefly described in the introduction of this thesis, Technical Debt (TD) is a metaphor for sub-optimal solutions [5]. The definition for TD 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” [4, 35]. Similarly to financial debt, it implies that there is a debt that must be paid. The cost of not repaying the debt in a system is associated with an interest. For as long as the debt is not repaid, the interest is expected to grow over time [3].

Reworking technical solutions and repaying the debt is often referred to as code refactoring, which as implied, is a reference to the process of restructuring code. The cost that comes from refactoring is regarded as the principal [34]. This reflects the cost of refactoring TD based on a certain amount of issues. For example, components that are tightly coupled, little to no documentation, or a proper test suite is lacking. Failing to refactor the TD will have implications for the software in the long term. This includes, for instance, difficulties when adding new features, and making it more costly to refactor in the future.

As it was described by its definition, TD can over some short period provide benefits for software development teams [30]. For example, if a software team chooses to implement solutions that aren’t optimal to have a faster time to market. I.e., a development team may find it favorable to take some TD so that they can deliver a business value faster. Although, as it also was explained, TD is seen as harmful when taking the medium- to long-term perspective into consideration. This has made developers believe that TD has to be actively managed [42].

Some software practitioners may not be aware of TD in their projects, and hence have no overview of it. Others may be aware of TD and therefore make notes about it (e.g. comments), which will then make it Self-Admitted Technical Debt (SATD) [42]. In any case, studies have shown that developers tend to discuss TD issues regardless of whether it is mentioned explicitly or not [37]. Occurrences, where developers may mention TD directly, can be through phrases such as “technical debt” or alike. Indirectly, developers may discuss TD without consistently mentioning any related terms.

Typical examples of TD can be found by inspecting code using different tools [27, 34, 45]. These tools, as was previously explained in the introduction of this thesis, will then analyze code for a set of known rules. Including such as code smells, the complexity of the code (cyclomatic complexity), coding standards, and more. For example, a tool might crawl a codebase, analyze the code and offer a complete report on the findings. However, despite a big selection of available tools that uses a wide variety of different approaches, existing techniques for estimating TD aren’t necessarily adequate to consider the overarching impact of it yet [29].

In this thesis, the estimation of TD is quantified by SATD expressed in developer discussions. To be more precise, discussions from OSS projects. I.e., there aren't any tools that are being used to estimate TD in the projects by looking directly at the code itself. Although, there does already exist a TD dataset for OSS projects that has analyzed a variety of OSS projects using SonarQube [25]. Instead, Natural Language Processing (NLP) is applied to developer discussions, so that discussions can be classified as having either TD references or not. Regardless of whether that is a direct reference to TD, or if the discussions don't seem to mention TD directly.

To be exact, developer discussions that are found in OSS projects on the platforms Jira and Github. The two will then be linked together through issue keys related to their issue tracking system, which includes all the discussions across both platforms. This also makes it so that the discussions and their relevant code can be related to specific parts of the project.

Exploring and quantifying this relationship between developers' discussions and TD issues may give rise to new ways of looking at TD prioritization. As Ozkaya et. al [37] concluded in their study, NLP can be useful for both further defining and understanding TD. In their study, NLP was used with Machine Learning (ML) to estimate the number of TD discussions from the Chromium OSS project [1]. Their work suggested that tracking TD through developers' discussions based on issue trackers, such as Jira and Github, can provide both an effective strategy for monitoring TD in large projects and TD prioritization. However, their results fell short of forming a basis of an oracle for this approach, concluding that there was a need for further refining the method. Including improving the accuracy for classifying TD issues, and improving the feature engineering. The latter is explained in the section for machine learning in the background of this thesis.

Furthermore, being able to quantify TD in this manner, may then give information that can provide to be an invaluable method for TD prioritization. As it may guide decision-making related to allocating resources, including decisions related to both the development of new features and refactoring of TD. Considering that the developer discussions may give a different view of TD in a project, by identifying design concerns that would otherwise be difficult to pin down. As opposed to code inspection, which simply inspects the quality of the code itself.

This could, for example, include discussions about architectural design concerns, suboptimal development choices that have been perceived as adhoc, and awareness of up-front solutions that are below par. In addition to any discussion about other concerns that are related to TD issues, which will then have the potential to accumulate interest in the project over time. Ultimately making it so that problems that could be hard to identify and track with code inspection alone, may be discovered.

[1]https://www.chromium.org/chromium-projects/

2.4 DevOps metrics

DevOps, also known as development and operations, addresses the relationship between software development (Dev) and operations (Ops). The intent behind it is to support a collaboration between the two so that continuous delivery of evolving software with high quality can be produced. Measuring DevOps processes, and hence the software delivery performance can be achieved with DevOps metrics. These are data points that are related to the development, delivery, and operations of the software.

Industry-standard has been to use the Four Key Metrics (FKM) to measure software delivery performance, as well as differentiate between software practitioners and organizations [16]. These metrics are identified by Google's DevOps Research and Assessment (DORA) [2] as:

- Deployment frequency
- Lead time for changes
- Change failure rate
- Time to restore service

While each metric will describe its aspect of producing and maintaining software. They can at a high level be split into two groups, one for velocity and the other for stability. The first two metrics, namely the deployment frequency and lead time for changes, belong to the first group. The last two, which are the change failure rate and time to restore service metrics, are placed in the stability group. In this thesis, the focus will be on the velocity metrics as these will measure the velocity of software development.

DORA has identified these metrics as good indicators for the performance of software development teams [16]. When they are used as indicators, development teams are placed into different performance tiers. These include the tiers: elite, high, medium, and low, where the elite performance tier is ranked as best and the low tier as the worst. This would then mean that they can be used to get insights into projects. Although not necessarily using the same ranking system entirely, but rather that the metrics are used with the OSS projects and their software delivery performance.

In this thesis, the metrics will be used in correlation with quantified TD issues that are classified based on SATD expressed in developer discussions. The purpose of this is to see if they can be used in an analysis, specifically as a way to get insight into how velocity correlates with TD. The basis of this is to further explore ideas from the study of Lenarduzzi et al. [28], which tried to use lead times as an estimate for the impact of TD. Coupled with how TD is known to affect performance measures, such as making it harder to add features and more costly to refactor.

[2]https://cloud.google.com/devops/state-of-devops

2.5 Natural language processing

For this thesis, Natural Language Processing (NLP) is used to quantify TD. This is a branch of Artificial Intelligence (AI) and linguistics, that concerns itself with trying to make computers understand human language [11]. As briefly explained in the past sections, the aim is to be able to use NLP on developer discussions to classify whether they are TD issues or not. To achieve this, NLP is combined with machine learning and an artificial neural network. Both of which are subfields of AI that will be explained in this section, as well as in the forthcoming parts of the thesis.

Quantifying the TD from developer discussions found in the OSS projects would mean that a large amount of data would have to be evaluated. It is because of this that NLP has been subsequently combined with machine learning as a classification method. As using traditional algorithms to classify the stochasticity of discussions could prove to be difficult. Considering that it may be difficult to predict the sentiment of a discussion, without the generalization that NLP and ML can provide in such cases.

Other studies, such as Maldonado et al. [48] and Zhongxin et al. [32] have shown that it is possible to detect SATD using NLP and ML. In the former, SATD comments from source code comments were detected using an NLP model from ten different OSS projects. This model would then analyze source code comments and decide if they expressed SATD or not. The latter on the other hand proposed a tool built on an NLP model that could be used for both text-mining and classifying SATD. Doing so by analyzing source code and flagging comments that contained SATD. They would then conclude in the latter study, that the method could be used to make developers aware of SATD comments in their source code.

Although both studies used NLP to detect SATD, they focused on source code comments. This is different from what Ozkaya et. al [37] did, as well as what this thesis tries to achieve. As source code comments may have a simple text format, whereas developer discussions may include more complicated problems to classify. Such as code snippets, comments from humans and bots, timestamps, and more. Furthermore, after the discussions have been classified they can be linked to their respective issues. Thus making it possible to quantify it as both TD issues and non-TD issues. The quantified TD can then be used together with performance metrics, which is in this case DevOps metrics, to see if it can bring insights into how TD is prioritized and fixed in projects.

2.5.1 Machine learning

To facilitate the language used for the Machine Learning (ML) for the forthcoming sections, this section will give the theoretical context for it. ML deals with getting computers to learn and improve through experience. This is achieved by building ML models on algorithms through a specified paradigm. After having been trained, an ML model will be capable of making decisions or predictions on its own. This makes ML a great tool for automating tasks that could otherwise be hard to define with traditional algorithms such as in the case of this thesis, where large amounts of discussions will be classified based on their sentiment.

The algorithms that ultimately make an ML model, will have to be trained using sensible input that is also known as training data. This data is chosen as a direct result of the paradigm that has been selected for the model, which can conventionally be divided into three categories. The first one is reinforcement learning, which is focused on rewarding the ML model if it gives the desired output, and vice versa. The second one is unsupervised learning, this paradigm tries to uncover hidden patterns on its own and subsequently gives an output without having humans interventing with the learning process. Lastly, there is supervised learning. This works by feeding the model with data that has already been marked so that the model can learn from this and apply what it has learned to unseen data. This thesis is concerned with the latter, as it will be used to train both ML models and an artificial neural networks for this thesis.

Further, another subject matter that is important for ML and NLP is the process of featuring engineering. This is concerned with trying to find relevant variables to use from the raw data (which has been collected from the OSS projects). More precisely, it involves using domain knowledge to systematically select useful features (e.g., attributes, properties, and so forth) directly from the raw data. As a result, the raw data can be turned into more useful data that can efficiently be used with supervised learning.

2.5.2 Artificial neural networks

While Artificial Neural Networks (ANN) is a subfield of ML, this thesis has decided to separate it from the other ML models, so as to not create any confusion. In contrast to more traditional ML algorithms, a neural network will try to mimic a human brain. In the case of this thesis, this has been achieved using deep learning and word embeddings. Both of which are explained thoroughly in the methodology chapter of this thesis.

The basic idea of a neural network is to mimic or simulate an artificial human brain, which it does through so-called neurons. These will make up a structure with multiple layers called a neural net. Making the neural network capable of taking input, doing calculations with the input, and then producing some output. Likewise to the other ML models, the ANN that is used for this thesis will be trained with supervised learning.

However, as opposed to classical ML models, the ANN will use deep learning to train. More specifically, supervised deep learning. This will together with the word embeddings, be able to leverage and automate the feature extraction from the dataset used for supervised learning. Thus, use the word embeddings to further learn without human supervision.

Chapter 3

Methodology

The purpose of this chapter is to describe and assess the systematic approach used to answer the research questions for this thesis. The first section summarizes the research process that has been followed. This process provides an overview of the research strategy and describes the practical steps that have been carried out. The next section describes the data collection process, which is used to extract data for both RQ1 and RQ2.

Lastly, there is one section for RQ1 and one for RQ2. These two sections present the methodology used for answering the RQs respectively. The results from this, as well as the analysis, are given in the results chapter and further discussed in the forthcoming sections. The raw data, scripts that were created and the final data have been made available for replication purposes and can be seen in appendix A.

3.1 Research process

The research process that has been adopted for this thesis is quantitative research. This is a deductive strategy that involves collecting and analyzing quantifiable data, so that the data may be used for empirical investigations. This includes mathematical models, statistical techniques and numerical analysis [18]. Applications for quantitative research involve testing for casual relationships, making predictions, finding patterns, and generalizing the findings to a wider population.

As this thesis aims to answer questions that can be quantified and analyzed in a numerical form, where such data is available from the OSS project, it was a natural choice to choose a quantitative research strategy. The approach is subsequently combined with a multiple-case study of the selected OSS projects, which can collectively give a greater understanding of the questions asked in the RQs. This is in contrast to single-case studies, which are more focused on depth whereas a multiple-case study is rather orientated towards breadth and diversity [20].

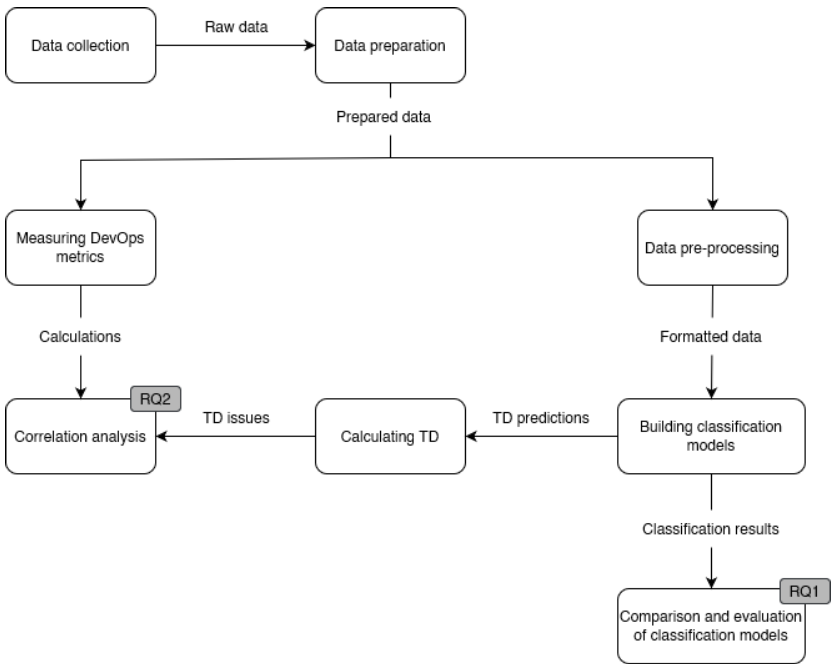


Figure 3.1: Diagram for the research process

The practical steps for the research process can be seen in the diagram in Fig. 3.1. The boxes represent the overall steps of the thesis and the arrows are the result of them. The steps that are performed to answer the research questions have their own marker.

Data collection: This step is explained in section 3.2 and addresses how relevant data is collected. Including the context for the data, how OSS projects are found, and ultimately how the data have been extracted.

Data preparation: As raw data from the data collection will contain a lot of unnecessary information, it will have to be cleaned, properly formatted, and prepared for use. The process for this is partly given in section 3.3.2 for the ML and neural network, and partly in 3.4 for the DevOps metrics. The results is summarized in section 4.1 of the results chapter.

Data pre-processing: Data pre-processing is an important process for the classification models. This will prepare the data specifically for NLP so that it will be correctly formatted and ready to be used with the ML models and neural network. The details for this is given in section 3.3.2.

Building classification models: This step presents how the ML models and the neural network have been built and used to classify issues. This can be seen in section 3.3.3, section 3.3.4, section 3.3.5 and section 3.3.6.

Comparison and evaluation of classification models: This step is performed to answer RQ1 and can be seen in section 4.2. In this step, the classification models will be evaluated based on their performance and results.

Calculating TD: This step involves calculating both the size of the classified TD issues and quantifying the TD for each project. The result of this will be a more precise measure that takes the size of the issues into account, thus resulting in better construct validity. The calculation for issue sizes can be seen in section 3.4.1 and calculation for TD in section 3.4.4. Calculating the TD will involve quantifying TD issues with their size as a final measure.

Measuring DevOps metrics: Measuring the velocity DevOps metrics includes calculating the lead time for changes seen in section 3.4.2, and deployment frequency seen in section 3.4.3. The size of the issues used for the calculations will also be considered, this can be seen in section 3.4.1.

Correlation analysis: This step answers RQ2 by correlating the quantified TD issues with DevOps metrics for insight into projects. As the correlation analysis will look at how TD can impact performance, the DevOps metrics will be correlated with open TD issues as they both evolve over time. The results from this can be seen in section 4.3.

3.2 Data collection

This section of the thesis addresses how data for RQ1 and RQ2 was collected from OSS projects. First, the context for how the data was collected is given. Secondly, the process for finding OSS projects is described. Lastly, the method for extracting data is presented. The results from the data collection have been summarized in the results chapter.

Collecting data from open source projects can be challenging. While there does exist a plethora of projects to choose from, each one may be significantly different from the next. For example, one project may use Jira for issue tracking, the other Bugzilla [1], and the third an entirely custom system. Some codebases may be available on Github, others may be hosted somewhere else. Even if two or more projects have chosen the same systems, they may have different layouts. For example, differences in how they define issue tracking types, categories used for prioritization, and so forth. The same goes for the SDLC that has been applied to each project, or in other words, the processes for how work is completed in the projects.

As it was described in the introduction of this thesis, the platforms Jira and Github have been chosen as sources for data. The reason for this is to simplify the data collection process so that it is possible to get a good overview of the projects. These are also platforms that are commonly used for OSS projects, which makes it straightforward to find projects that use them. The collected data will therefore include all the information that can

[1]https://www.bugzilla.org/

be collected from issues in Jira. For Github, all the associated pull requests (PRs) connected to the issues will be collected. This will make it possible to relate issues directly to relevant code and be used for the DevOps metrics.

For Github, it can be pointed out that some projects may require that a standardized template is filled out for PRs, before any merging into their codebase, also known as a repository, can be accepted. This process is an event that happens when developers want to merge code into a codebase on so-called branches, which in short, will represent a contained line of development that is independent of the main repository. The latter is usually called the “Main” branch. Other branches may be used for specific reasons, for example, testing different versions of a software product.

Moreover, in the open source community it is common practice to perform a pull request, then test and review it before changes to a branch are accepted [49]. As a result of this, contributors can do a so-called fork of the branch from a project they want to make changes to, which will take a copy of the original branch. From there, the contributors can safely make changes to the forked copy and ultimately a pull request (PR). After the PR is created, it will most likely have to pass two steps. The first one is some sort of automatic testing, which will test the code in the PR to see if it is compliant with the rest of the code in that branch. The second step is usually that a reviewer will review the code as a form of quality assurance (QA).

As briefly mentioned, some OSS projects may practice a formal procedure for accepting PRs. Others may practice a less formal and more flexible method for accepting merges. Whatever is the case, the PRs will usually be linked to their respective issue tracking. This makes it so that the code and conversations are linked and don’t have to be tracked separately. For example, given any Jira issue, which is usually in the format of a unique key like “KEY-1234”, all associated PRs may be related to the issue based on the key. Thus, making the data relational on basis of the key.

This is something that had to be taken into consideration when data from OSS projects were collected so that the whole perspective of an issue could be considered. Only parts of a conversation may be located in a Jira issue, while the rest of the conversation may be found in the comments of a PR along with its code. In addition to this, being able to track the code will also make it possible to collect quantitative data that is specifically related to software delivery performance. E.g., the size of the code in the PR, when the code was merged, which branch it was merged into, and so forth.

Furthermore, when it comes to selecting the OSS projects, certain requirements have been given. First of all, the projects must be of a significant size both in terms of their codebase and management system. Secondly, the open source projects must span several years and have been reasonably active across that period. Lastly, the projects must have an open codebase and management system. All in all, making it so that there’s a possibility that technical debt may have been accumulated in the project.

Moreover, when it comes to the actual data collection, creating a tool for this may be difficult, considering that there are likely to be differences between the projects. However, there will usually be certain standards and attributes that overlap between the different systems, such as textual data-exchange file formats that are both readable for humans and machines. For example, a platform like Jira will support the extraction of labeled data from issues in a comma-separated values (CSV), RSS, or XML format. Github on the other hand has a REST API that can be interacted with. This makes it so that API requests that return data formatted as Javascript object notation (JSON) can be created.

Having these standards makes it possible to extract both quantitative- and qualitative data from the different open source projects. For example, commit sizes from Github and conversations out from comments on Jira issues respectively. Even if the information is spread across different tools and systems, such as Jira and Github, it will be possible to join the data based on common data points (i.e. the issue keys).

This is in contrast to something like having to use very customized web scraping to obtain useful information. This may be the case for custom management systems, as well as systems that purposely make it hard to scrape data from them. For the latter, everything would have to be specified, for example through a plethora of different scripts, where any small difference has to be facilitated, which could be a time-consuming task.

3.2.1 Finding projects

Different online searches for "open source projects" revealed a great number of projects that all met the requirements. Many had been active for years, consisted of codebases with millions of Lines Of Code (LOC), and had issue tracking systems with thousands of issues. However, to limit the number of projects, they were selected based on structural similarities and whether they had any SATD. The structural similarities ended up with them all following an equal issue key system, where issues would be relationally based on the key. Thus making it possible to track the individual issues to PRs.

The SATD that was identified was found to be 320 Jira issues that had been explicitly marked as TD. They were spread across the different projects, where the developers themselves had marked the small subset of issues with TD as the issue type. Although this is a small number compared to the total amount of issues, it could later be used to train the classification models with supervised learning. Beyond simply the structural similarities, the projects were selected based on both technical- and organizational variety. This is in consideration of the external validity of the thesis, as the projects would then represent a broad range of diversity. Including different SDLCs, programming languages, frameworks, size of codebases, and amount of contributors, as well as different purposes for the projects.

This made it possible to narrow it down to five different projects, which seemed to be based almost exclusively on Github and Jira. Thus collecting data for each project would also be consistent and not require drastically different methods, such as customized web scraping. Table 3.1 has summarized all of the OSS projects that were selected in this thesis.

Name	**LOC**	**Pull-requests**	**Jira issues**	**Contributors**
Beam	1,062,474	16,137	13,436	850
Flink	1,837,364	18,031	25,225	979
Sakai	1,234,162	9,684	43,306	213
Wildfly	817,550	14,897	12,947	334
WiredTiger	196,364	6,284	8,282	57
Sum	**5,147,914**	**65,033**	**103,147**	**2,433**

Table 3.1: Open source projects

Wildfly is a Java-based application server that is formerly known as JBoss. It is cross-platform and implements a set of specifications that extends the Java platform with enterprise features [2]. Beam, also known as Apache Beam, is on the other handwritten primarily in the programming languages Go, Python and Java. It is software that is made for both defining and executing data processing pipelines, which in short, will enable automated processes used to assist developers with compiling, building, and deploying their software products [3].

Flink, also known as Apache Flink, is a framework written primarily in Java and Scala. It is a processing engine that is built for stateful computations and data streams for both unbounded and bounded data. In essence, this means that Flink can process huge amounts of data fast and reliably in real-time [4]. Sakai is an extensive educational software platform that has a wide variety of features and is primarily written in Java [5]. Lastly, there is WiredTiger, which is a NoSQL data management platform that is primarily written in the C programming language [6].

These projects are wide-ranging and feature different technical and organizational varieties. Including such as different SDLCs, a great vary in different sizes, having been written in different programming languages and serving different purposes. For example, the WiredTiger project is very different and rather small compared to Flink. Whereas the former is around 196,000 LOC, has had 57 contributors, is written in C, and is meant to serve as a NoSQL data management platform. The latter on the other hand is a Java-based framework that has almost 10 times the amount of LOC and is closer to 1,000 contributors than 100.

[2]https://www.wildfly.org/
[3]https://beam.apache.org/
[4]https://flink.apache.org/
[5]https://www.sakailms.org/
[6]https://www.mongodb.com/docs/manual/core/wiredtiger/

3.2.2 Data extraction

Extracting data from both Jira and Github was done in multiple steps. An activity diagram in Fig. 3.2 describes the process. The final result from the data collection can be seen in the results chapter.

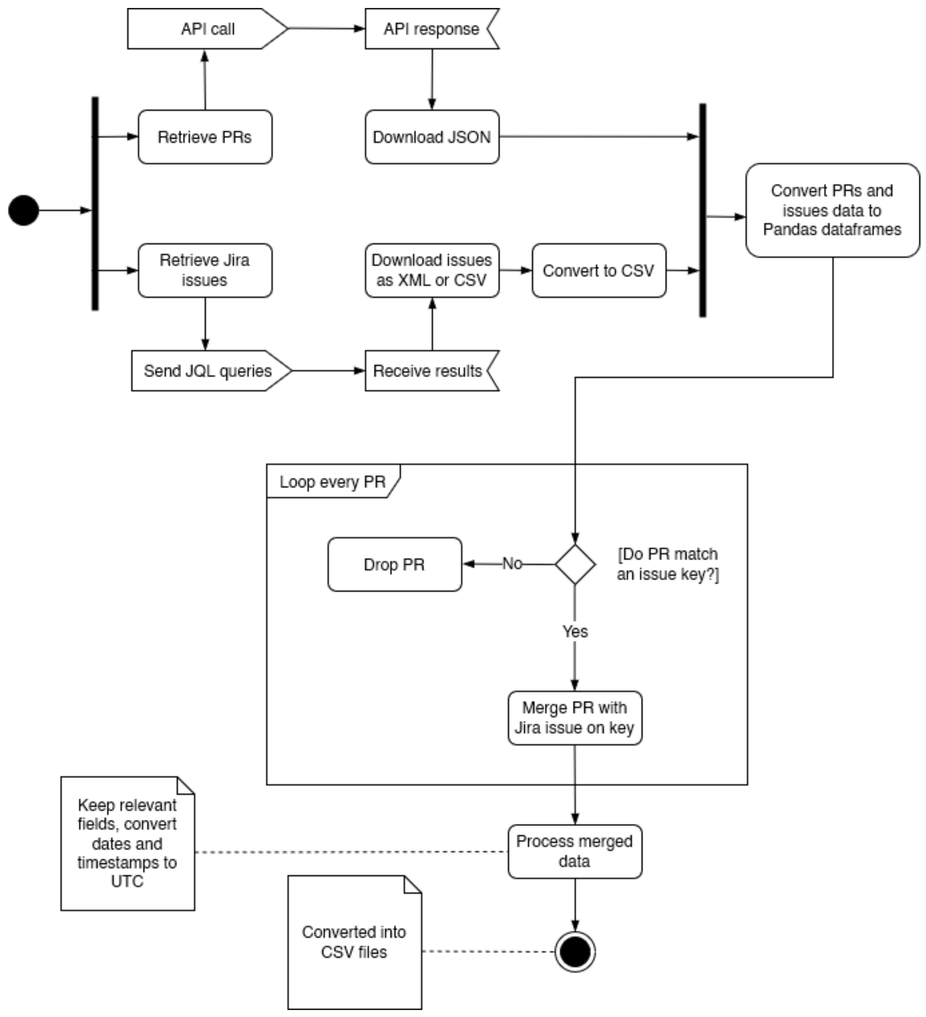

API call
API response
Retrieve PRs
Download JSON
Convert PRs and issues data to Pandas dataframes
Retrieve Jira issues
Download issues as XML or CSV
Convert to CSV
Send JQL queries
Receive results
Loop every PR
Drop PR
No
[Do PR match an issue key?]
Yes
Merge PR with Jira issue on key
Keep relevant fields, convert dates and timestamps to UTC
Process merged data
Converted into CSV files

Figure 3.2: Activity diagram of data extraction process.

The exporting of all Jira issues from the different projects was completed using the Jira Query Language (JQL) [7], then downloading and storing the data as either CSV or XML (depending on what the project allowed). For Github, the data from the associated PRs was fetched using API requests to the Github REST API [8], which was completed using the Python programming language and PyGitHub library [9]. Lastly, the processing and merging of data were completed using Python and the Pandas library [10].

The JQL made it possible to formulate query strings that were used to customize advanced searches in Jira. Each search would then allow for adding filters and specifying attributes for the different issues, which were then later exported. However, due to performance concerns in regards to such as memory exceptions, Jira would by default limit each export to the first 1,000 issues only. Unless the administrators of the project had explicitly changed the configuration themselves, which most had not.

This caused a limitation that had to be circumvented for all the issues to be exported, wherewith the use of JQL it was possible to add a start- and end parameter to the index values of each search. This made it so that all the issues could be exported in chunks of 1,000 issues, rather than the first 1,000 only. However, this was not the only problem with extracting data from Jira. Some projects would also limit the alternatives for export formats, making it not possible to select CSV. In that case, other formats were chosen, such as XML, and then later converted into CSV. Thus all the Jira data would end up being homogeneous and therefore easier to process.

The PR data from each project's Github repository was retrieved by sending GET requests to the Github REST API. This was achieved by writing a Python script that used the PyGitHub library, making it straightforward to make the requests, specify its parameters, and retrieve data for each PR. The possible parameters could then include everything from PR creation- and closed timestamps, the state of the PR, commit details, and so on. It was therefore easy to select specific values and ignore irrelevant details about all the PRs that are associated with each project.

However, one challenge with Github was that its API had a rate limit, which only allowed 60 requests per hour for unauthenticated requests. This was not sufficient for collecting all the project's PRs. To increase the limit, the API requests had to be authenticated with basic authentication or OAuth. Using a basic authentication, it was possible to make 5,000 requests per hour for each repository. With a Github Enterprise Cloud account, it was possible to make up to 15,000 requests per hour.

[7]https://www.atlassian.com/blog/jira-software/jql-the-most-flexible-way-to-search-jira-14
[8]https://docs.github.com/en/rest
[9]https://github.com/PyGithub/PyGithub
[10]https://pandas.pydata.org/

The next step after that was to process all the data and then finally merge it. In this process relevant fields would be kept, dates and timestamps had to be converted into coordinated universal time (UTC), and redundant data excluded. The processing was completed using Python and the Pandas library, where the Github data would be parsed from JSON and Jira from CSV formats into Pandas data frames. This data type is a two-dimensional data structure that is size mutable, making it easy to both manipulate and reshape data on structured sets like CSV and JSON [36].

Merging the pull requests with Jira data was done by comparing their respective key labels with each other. With a regular expression (regex), one could match the key field on a pull request with the key in a Jira issue to determine whether they belonged together or not. For example, if a Jira issue had the key “SAK-45531” and a Github pull request had the very same, the Jira and Github data would then be merged based on the relation the key established. The result, seen as records, would then be:

```
{
  0: [
  {
  "title": "[BEAM-5759] ConcurrentMo...checkpoint finalization",
  "description": "When reading from a JmsI...ception will be thrown.",
  "key": "BEAM-5759",
  "type": "bug",
  "status": "Resolved",
  "resolution": "Fixed",
  "created": "Tue, 16 Oct 2018 10:53:35 +0000",
  "resolved": "Wed, 17 Oct 2018 13:51:53 +0000",
  "comment_0": "FAO jbonofre - the corre...sue (as well as a fix).",
  "comment_1": "Thanks for catching. I'm  reviewing the PR.",
  ...
  "github_number": 6702,
  "github_state": "Closed",
  "github_title": "[BEAM-5759] Ensuring Jms...sed and modified safely",
  "github_body": "As described in [BEAM-57... | --- | ---",
  "github_base_ref": "master",
  "github_created_at": "Tuesday Oct 16, 2018 at 11:15 UTC",
  "github_updated_at": "Wednesday Oct 17, 2018 at 13:51 UTC",
  "github_closed_at": "Wednesday Oct 17, 2018 at 13:51 UTC",
  "github_merged_at": "Wednesday Oct 17, 2018 at 13:51 UTC"
  }
  ],
  1: [
    {
    ...
    }
  ],
  ...
}
```

3.3 Detecting TD discussions

This section deals with how TD can be quantified by being detected in developer discussions. As briefly explained in the introduction of this thesis, the OSS project discussions are taken from both Github and Jira. Further, classifying the discussions as either TD- or non-TD discussions has been completed using NLP. This has been achieved with supervised learning for Machine Learning (ML) and a neural network. To be more precise, an Artificial Neural Network (ANN) and two other ML models.

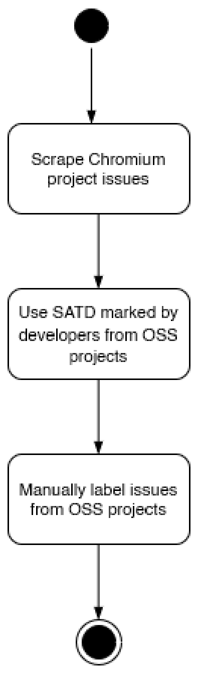
In order to train all three classification models, a dataset for supervised learning had to be separately collected and structured with both labeled training- and testing data. After training the models using the dataset with supervised training, the classification models were able to make classifications on the unseen data from the OSS projects.

The two different ML models that have been used are the multinomial Naïve Bayes and logistic regression. The ANN is a Recurrent Neural Network (RNN), which is one type of ANN. All of them are used for sentiment analysis on the text found in the data that was previously collected from the projects. The training- and testing data, as well as the data from the OSS projects, had to go through a data cleaning process in order for the models to make classifications. This process, along with how both the training- and testing data have been constructed, is explained in this chapter.

Results from the ML models and RNN can be seen in the results chapter of this thesis. This chapter includes the classification reports for each of the models, details that influenced the classifications, as well as the results from the predictions. It is based on this information that a classification model is chosen to classify issues as either TD or non-TD from the OSS projects data, thus quantifying TD issues from the developer discussions. The predictions from the classification model are then subsequently used with the DevOps metrics for RQ2.

3.3.1 Dataset for supervised learning

This section describes how a final dataset, which is used for supervised learning, has been constructed based on three steps. An activity diagram that summarizes the steps can be seen in Fig. 3.3. In the first two steps, the data is entirely collected from external sources. These two steps are referred to as the initial dataset. Data from the third step has been based on manually labeled data from the OSS projects for this thesis. The data from all three steps added together is referred to as the final dataset.



Scrape Chromium project issues
Use SATD marked by developers from OSS projects
Manually label issues from OSS projects

Figure 3.3: Process for creating the dataset used for supervised learning.

The reason for distinguishing between the different steps is that both the initial- and final dataset has been tested separately. In the initial dataset, as just briefly mentioned, the data has been collected from two separate and external sources. In other words, this dataset isn't affected by the subjective judgment from this thesis, but rather by the sources it originates from. The third step, however, is based on data that has been manually labeled specifically for this thesis.

Using the initial dataset simply on its own, didn't seem to give sufficient results for the classification models. It was therefore decided that the initial dataset had to be extended with more manually labeled issues from the OSS projects. This process, including both how the initial- and final dataset has been constructed, will be thoroughly explained in this section. The results from using both the initial- and final datasets, can be seen in the results chapter of this thesis.

The initial dataset that was used for training- and testing data was built up using a study from Ozkaya et al. [37]. In this study, a total of 1,934 TD references had been manually labeled from the Chromium open source project. The labels, which were published as a list along with the study, were selected by expert raters that had used a rubric from Bellomo et al. [8], seen in Fig. 3.4. This recognizes TD in project discussions, where they have not been explicitly highlighted. For example, by not having been mentioned with words such as “technical debt”, but may instead have been characterized by how the developers have expressed concerns related to TD.

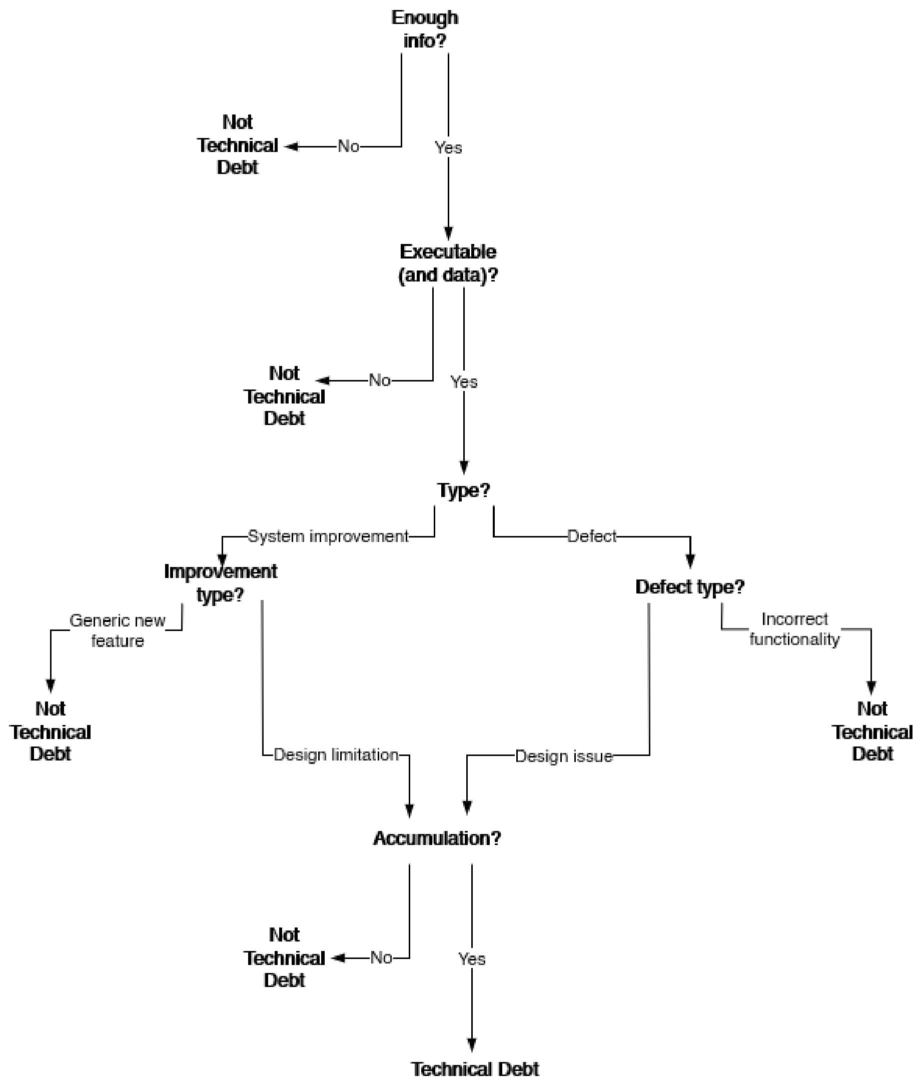


Figure 3.4: Reprinted rubric from Bellomo et al. [8] on how to identify and classify TD discussions in a system.

As seen in Fig. 3.4, the rubric is based on a decision tree that helps with separating TD issues from other things like user stories, tasks, defects, and new features. As issues themselves don't have to be explicitly marked or highlighted with any TD labels. The first question in the decision tree is whether the issue contains enough information for a decision to be taken. If not, then the issue isn't seen as a TD issue. However, if there is enough information, then the next question will ask whether it addresses an executable system artifact (e.g., code, scripts, tests, etc.), which is compared to non-executable artifacts (documentation, policy issues, etc.), or if it is a data problem (e.g., poorly designed database architecture).

If it does not, then it is not seen as a TD issue. However, if the answer is yes, then the rubric will move on to the next question. Here, the question will ask if the type is simply a defect or system improvement. At this point, the tree will now split up into two parts based on the types. If the type is a defect, the follow-up question will be whether it is an incorrect functionality or a design issue. Whereas the former is not seen as a TD issue, the latter is seen as a TD issue only if the design issue has been accumulated. In other words, if the design issue is causing work that is unintended, then this may increase the time to deliver.

The last path in the decision tree is if the type is a system improvement. Here, the follow-up questions will be about what type of improvement it is, which can either be a design limitation or a generic new feature. The latter is not seen as a TD issue. However, if the design limitation is an accumulation, then it will be seen as a TD issue. In this case, an accumulation could mean that something is hindering the ability to add new features in a reasonable time. Another reason could be that the current state, due to the design limitation, does not support improvements.

Further, as far as this thesis knows, Ozkaya et al. [37] did only publish the labels for the TD references. It was therefore necessary to collect the relevant data from the Chromium project based on the labels, so that this may be used for the dataset and supervised learning. In this case, each label that was manually classified by an expert rater would also be associated with a unique number that was linked to a single Chromium issue. Thus making it possible to connect the classification with the data, where the data would then be associated with its corresponding rating. This made it possible to use the labels and scrape the data from the Chromium project website, although it was a time-consuming process.

The labels from the study had originally been classified probabilistically, where they would fall in a range from 0 to 1. Distinguishing between different indications of TD found in the discussions. For example, a rating of 0.2 would mean that there are small hints, whereas 0.7 would mean that there are substantial indications present. This is a slight deviation from the rubric, which only encodes binary values, where 0 would mean "definitely not TD" and a 1 would mean "definitely included TD". In this thesis, however, as it is with the rubric, classifications will be either a 0 or 1. This will make it so that 0 can be used to indicate "not TD" and 1 as "TD".

The tool that was chosen for the web scraping task was Selenium WebDriver with Python. This is a tool from the open source project Selenium [11], which is a popular tool for test automation for web applications [19] and for web scraping [52]. The reason for its popularity is that it will drive a web browser, just like any user would, natively. This makes it possible to control the user agent, as well as both discover and manipulate DOM elements [12] inside the web content. In this case, that meant scraping Shadow DOM [13] elements that were loaded dynamically with Javascript.

The data that was ultimately scraped was based on feature engineering carried out by Ozkaya et al. [37]. In their study, a whole range of different types of features were both generated and tested for, although only a few turned out to be useful. The features that gave the most information gain turned out to be word vectors generated by free text, key phrases, and counts associated with the issues. Rather than information such as metadata (e.g., the author's email or issue status) from the Chromium issues.

This information helped with concluding what type of data would ultimately be scraped from each Chromium issue, which ended up being the issue fields: title, description, and all related comments. All of these fields could then be merged as a singular free text, that could later be turned into word vectors. The free text would exclude metadata and noise, such as the names or dates related to the issue, symbols (e.g., newline or address signs), and URLs. The reason for this is that noisy data is meaningless information that is not likely to be useful [1].

Since the rubric from Bellomo et al. [8] was used, the number of issues that had to be scraped was reduced. In that only classifications of either 0 and 1 were kept, and that anything between that range was discarded. This decreased the number of TD references from 1,934 down to 1,487. After writing a Selenium script that scraped each page for the issues on the Chromium project, a total of 1477 issues were collected. The result from the total issues was that 360 (24.37%) issues had a score of 1 (i.e., definitely included a TD discussion), and 1117 labels (75.63%) issues had a score of 0 (i.e., definitely not a TD discussion).

In addition to the issues that were scraped from the Chromium project, an additional amount of 320 TD references was also added to the initial dataset. As briefly mentioned in the introduction of this section and the data collection. These were issues that had been explicitly labeled by the developers themselves. In this case, out of the 320 issues that were marked as TD from the projects, only 162 issues were found to be useful. To be exact, 84 were from WiredTiger and 78 from Flink.

[11]https://www.selenium.dev/
[12]https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model
[13]https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM

The final result for the initial dataset was a total of 1,639 TD issues, where 1,117 (68.15%) were non-TD references and 522 (31.85%) TD. The average length (arithmetic mean) of a TD issue was roughly 973 characters, and for non-TD issues it was 1,185. Fig. 3.5 represents the distribution.

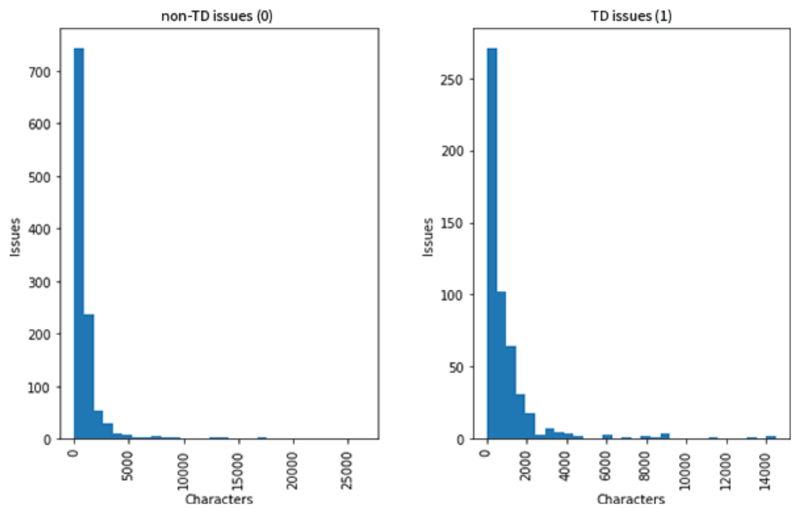


Figure 3.5: Length of text per issue in the initial dataset

However, the initial dataset was found to be too limiting for the NLP. As when the dataset was used for supervised learning with the two ML models and neural network, the predictions didn't seem to generalize adequately. This means that the ML models and the neural network did not adapt properly to data that haven't been seen before. In this case, the prediction results from running the models and neural network on the OSS projects would give fluctuated results with big differences between all the different projects. In which the classifications of TD issues would range from as little as 11.96% to as much as 76.47%.

Although the ML models and neural network, after having been trained with the initial dataset, were able to some extent learn and generalize. The data from the initial dataset didn't seem to be applicable when taking into account that the OSS projects are different from each other. Considering that they all differ in both technical- and organizational variety, where some may differ significantly more than others. For example, the Chromium project is primarily written in the C++ programming language, Wildfly and Flink on the other hand, are written in Java.

In consideration of the above, new labels were marked and added to the initial dataset. This way the dataset, used for both training and testing the ML models and neural network, would not just contain data from the Chromium project alone, but would also feature data from all the OSS projects themselves. This would make up for the potential differences across all the projects. The summarization of this can be seen in Table 3.2.

Project	**# of labels**
Beam	150
Chromium	1,477
Flink	228 (78 by developers)
Sakai	150
Wildfly	150
WiredTiger	334 (84 by developers)
Sum	**2,189**

Table 3.2: New dataset with manually marked issues

As seen from Table 3.2, the new dataset that extends the initial dataset ended up with a total of 2,189 issues. This includes the issues labeled by developers and the data from the Chromium project, as well as the new issues from the OSS projects themselves. The issues from the OSS projects have been manually marked for this thesis using the rubric from Bellomo et al. [8], which was previously explained in this section.

Fig. 3.6 represents the distribution of the final dataset. This dataset gives a significant increase in the average length of a TD issue, with 4,261 characters compared to 973 for the initial one. For non-TD issues it is 7,540, compared to 1,185 for the initial dataset. The increase is likely due to more information (Jira and Github) being collected from the OSS projects.

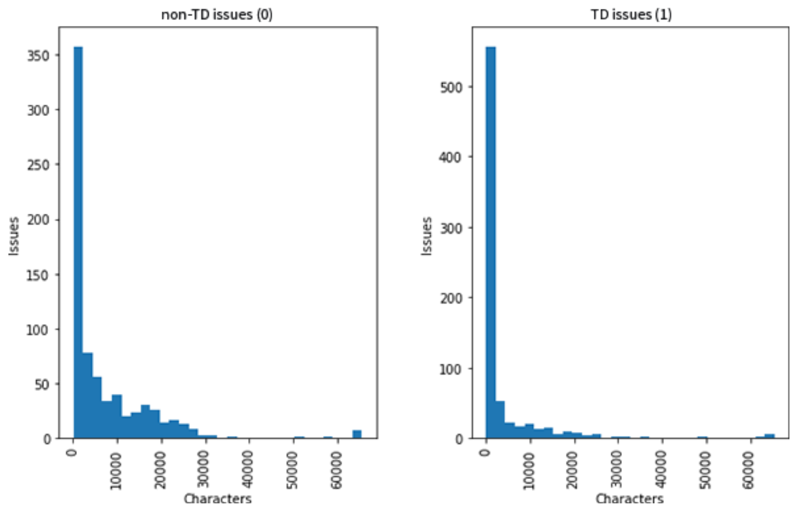


Figure 3.6: Length of text per issue in the final dataset has been increased compared to the initial dataset.

3.3.2 Data cleaning

Cleaning the dataset

Some of the necessary data cleaning for the dataset had been achieved during the collection processes. When the Selenium script was used, it removed HTML tags from the data. Although code snippets from the discussions themselves were an exception to this, as they will be used for the sentiment analysis. Further work would then include removing unnecessary symbols and stop words from the English language. In addition to this, text normalization will be applied to the data. Lastly, the data will have to be checked and verified for any corruption or incorrect formatting.

The reason for cleaning the data, especially for NLP tasks, is to minimize the clustering of words and maximize what one can obtain out of the data. In that, the raw text in itself can be hard for machines to understand completely on their own. Consequently, one would include the process of data cleaning to assist the machines with “understanding” better. Furthermore, it can also be pointed out that NLP tasks can be very different from one another. In this case, the task is to analyze development discussions, which can be different from other typical NLP tasks like smart assistants or email filters. The process of data cleaning may therefore also differ from one task to another.

The data cleaning process was completed with Python using Pandas, Numpy [51] and Natural Language Toolkit (NLTK) [33]. Here, Numpy and Pandas were used for pre-processing of the data, specifically in regards to data manipulating and modification. For example, such as removing missing values (e.g., NaN) in the data and merging data. NLTK's corpus was used for removing stopwords from the English language, in addition to text normalization through stemming and lemmatization. A section of the raw data from the dataset can be seen in Table 3.3.

issue	text	TD
367158	['Issue 367158: Simplify EncryptedMediaIsTypeSupported* tests.', 'Currently...	1
445880	['Issue 445880: Parallelize test execution to speed up buildbot runs', 'pbos@ h...	1
490895	['Issue 490895: Huge animated GIFs can lead to scroll jank. UserAgent: Moz...	0
...	...	...

Table 3.3: Section of issues with raw data

The data cleaning process that has been used for the dataset is:

- Removal of redundant key tags from the issue.
- Removal of unnecessary symbols.
- Normalization through stemming and lemmatization.
- Lowercasing all text.

Removing unnecessary symbols (e.g., newline symbol `\n`) and redundant key tags (e.g., a ['Issue 367158'] tag) from the text is fairly easy with the use of regex functions in Python. It involves finding patterns that match the symbol or key tag marks. When found, these can then be removed by replacing them with nothing, which will ultimately clean up the text. For example, a sentence like "['Issue 367158: Hello! How are you?`\n`" will translate into "Hello ! How are you ?". Even though this may make it confusing for humans, considering that certain symbols and tags may be needed for a text to be properly understood. It may help classifiers as the raw text will now be cleaned up and have the noise removed [21].

Text normalization, on the other hand, will in this case include the concepts of stemming and lemmatizing the text. The purpose behind applying these is to standardize the text, as well as decrease any randomness in the text. In this case, the text normalization is achieved by stemming a word's lemma. This is done by using linguistic rules (achieved through regexes) to try to find the root form of a word (lemmatization). Then, the words will be derivationally reduced into their common base form (stemming).

For example, the stemming of the words "talks", "talking" and "talked" will simply derive into "talk". Lemmatization is slightly more complicated, as it addresses the process of using morphological analysis. Before it removes the inflectional form of the word and transforms it into its root form. This includes such as using the vocabulary, grammar relations, and structure of a word. For example, the words "is", "are" and "am" may simply be derived into "be". The final word "be" will then be a so-called lemma.

A snippet of the final dataset, after having gone through the process of data cleaning, can be seen in Table 3.4.

issue	**text**	**TD**
367158	simplify encryptedmediaistypesupported * test .. currently lot duplicate...	1
445880	parallelize test execution speed buildbot run . pbos @ excellent work...	1
490895	huge animated gifs lead scroll jank . useragent : mozilla/5.0...	0
...	...	...

Table 3.4: Section of issues with cleaned data

Cleaning data from the OSS projects

Likewise to the dataset, the data from the actual projects have to be cleaned as well. That way the data from the projects can be correctly interpreted by an ML model or neural network. Then, subsequently, be used to make classifications on whether the issues from the projects are TD issues or not. Cleaning the data will make the text from the projects match the same format as the dataset that was used for learning how to predict.

Considering that the process behind collecting the data was different, as well as the data itself is different, the process of cleaning the data from the projects will be slightly different compared to the dataset. Furthermore, because some of the data from the projects were used for constructing the dataset. Namely, the issues that were marked by the developers as "Technical Debt" issues. Then these will also have to be removed so that they won't affect any classification results from the ML models or neural network, when they are trying to perform a prediction.

The data cleaning process that has been used for the projects are:

- Removal of issues marked as "Technical Debt" by developers.
- Merge and combine all comments related to an issue into a singular issue, along with the issue title and description.
- Merge all the Jira and Github text data into a single column.
- Removal of redundant key tags from the issue.
- Removal of unnecessary symbols.
- Normalization through stemming and lemmatization.
- Lowercasing all text.

3.3.3 Multinomial Naïve Bayes

There are lots of ML models that exist and that can be applied to NLP problems. However, as a baseline model, the multinomial Naïve Bayes classifier model has been selected. This is a model that is simple and probabilistic, which considers each feature as both equal and independent. Although a simple model, it has been shown to perform well on NLP problems [22].

The multinomial Naïve Bayes classifier builds on Bayes' theorem, which is used to calculate conditional probabilities [10]. As seen from the Eq. 3.1, the theorem calculates the probability of some event A occurring given that B has already occurred. This is done by calculating the probability of B occurring given that A has already occurred. Then, multiply that with the probability of A occurring. Finally, the expression will be divided by the probability of B occurring.

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{3.1}$$

This mathematical formula describes its conditional probability. I.e., the probability of some event occurring will be conditionally dependent on another event that has occurred. This makes it possible to find probabilities if certain other probabilities are already known. Whether that is by evidence, presumption, assertion, or assumption. Ultimately making it so that it can be said that A will occur given that B happens.

There are three types of classifiers that are Naïve Bayes, namely the Bernoulli-, Gaussian- and Multinomial Naïve Bayes classifiers. In this case, the latter has been used because one of its main applications is document classification. This classifier does so by generating a multinomial distribution of the different events, which can be done by representing feature vectors as the frequencies of the events.

Specifically, given that a class variable x represents a set of n features as $x = (x_1, x_2, x_3, x_4, ..., x_n)$. Then the probability of $P(y|x_1, ..., x_n)$ can be calculated using the chain rule, as expressed in Eq. 3.2.

$$P(y|x_1, ..., x_n) = \frac{P(x_1|y)...P(x_n|y) \cdot P(y)}{P(x_1)...P(x_n)} \tag{3.2}$$

Furthermore, what makes this naïve is assumed conditional independence. Particularly, there are mutual independence for all the n features in x. This will make the denominator remain static (i.e., it won't change for the entries). The distribution can be expressed as seen in Eq. 3.3, where proportionality (denoted by $\propto$) is inserted and the denominator replaced.

$$P(y|x_1, ..., x_n) \propto P(y) \prod_{i=1}^{n} P(x_i|y) \quad (3.3)$$

For example, given a very simplified dataset D, with a binary set of classes yes and no, which indicates either a TD discussion (being yes) or non-TD discussion (as no). Where D has three independent features (thus being naïve), which are f_1, f_2 and f_3. The posterior probability of $P(yes|x)$ would be calculated as $P(yes|x) = P(f_1|yes) \cdot P(f_2|yes) \cdot P(f_3|yes)$, while for the probability of $P(no|x)$ it would be $P(no|x) = P(f_1|no) \cdot P(f_2|no) \cdot P(f_3|no)$.

This makes it possible to use the multinomial Naïve Bayes classifier for the classification of text problems, because of how each word in a sentence can be represented as an independent feature. Making it so that each of the single words in a sentence will be used, rather than the whole sentence itself. The probability of a word occurring together with another will therefore be based on their individual probabilities.

Furthermore, in order to make this applicable to text problems. The text has to be translated into something that can be understood by computers, which is numerical values. One way of doing this is to assign a unique number to each word in a sentence so that the number will represent the word. The words can then be presented as a matrix with all the counts (also known as a bag of words). An example can be seen in the pseudocode:

```
corpus = ['this needs an update',
          'an update was postponed',
          'this should require an update']
>>> X = vectorizer.fit_transform(corpus)
>>> print(X.toarray())
[[1 1 1 1 0 0 0 0]
 [0 0 1 1 1 1 0 0]
 [1 0 1 1 0 0 1 1]]
```

In this case, the pseudocode is based on the scikit-learn library [39]. From the pseudocode, it can be seen that a corpus with three sentences has been fed to a text vectorizer. There, a method that learns the vocabulary from the corpus and then returns the document-term matrix is used. I.e., each word is represented in the matrix as a feature and the dimensionality is equal to the vocabulary. In such a way each unique word can be understood as columns and the full text as the rows.

The scikit-learn library is also used for the implementation of a multinomial Naïve Bayes classifier in Python, where the foundation is built on the same processes as explained here. This is then followed by the use of the term frequency-inverse document frequency (TF-IDF), which is further explained in this thesis. Finally, all of it is used for both model training and testing. The latter is further explained in the results section of this thesis.

Making a prediction with the Naïve Bayes classifier can be done using the joint probability $p(x, y)$ of the class and data, as expressed in Eq. 3.4. This makes it a generative model, considering that it will need to model how the data was generated before it can start making predictions. Particularly, it will have to learn the distribution of data before it is able to generate data instances that are new.

$$\operatorname{argmax} P(x, y) = \operatorname{argmax} P(y) \cdot P(x|y) \tag{3.4}$$

As mentioned before, the implementation of the Naïve Bayes classifier was done using the scikit-learn library with Python. The results from using the classifier can be seen in the results section of this thesis. Here, both the training and test data from the manually labeled dataset, as well as the data from the projects have been used with the classifier.

3.3.4 Term frequency–inverse document frequency

Term frequency-inverse document frequency (TF-IDF) is a statistical measure used to calculate the relevance of a term (word) in a corpus (collection of texts). This can be achieved by multiplying the term frequency (TF) with the inverse document frequency (IDF). The TF can be expressed as it is seen in Eq. 3.5, and the IDF as seen in Eq. 3.6.

$$tf(t, d) = \frac{\text{number of occurences of } t \text{ in } d}{\text{number of total words in } d} \tag{3.5}$$

$$idf(t, D) = \log\left(\frac{\text{number of documents}}{\text{number of documents with } t}\right) \tag{3.6}$$

Here, D will represent the entire corpus, d a document inside the corpus, and t the term. The TF metric will return a value based on the frequency with which the term occurs in the document. The higher the value, the more frequently the term appears in a document. The IDF on the other hand will calculate a value that is based on how rare or common a term is for the entire corpus. The logarithmic scale in IDF makes it so that the proportionality of the frequency of a term does not grow with the relevance.

$$tfidf(t, d, D) = tf(t, d) \cdot idf(t, D) \tag{3.7}$$

The reason behind multiplying these two metrics together, as seen in Eq. 3.7. Is that including only TF would not necessarily say much about the relevance of a term (word) in a text. Considering that the terms which are occurring the most in the text would also be those with the highest term frequency. This could then for example include stopwords, hence why they were removed in the data cleaning, as well as other casual words that shouldn't necessarily have any great importance.

Instead, the TF metric may be multiplied by the IDF. This will help determine how important the terms are, rather than how frequently they appear. The final result will then be a TF-IDF score, which is one way to help determine the overall relevance of a term in relation to a corpus. The higher the TF-IDF score, the more relevant the term would be.

This has then been applied to the data collected in this thesis. First, a text vectorizer was used on a corpus so that the text can be transformed into numerical values. In this case, the data was the actual corpus. Then, the TF-IDF score was calculated and used together with the ML models. For this thesis, the selected models were the Multinomial Naïve Bayes and a logistic regression model. The latter is further discussed in this chapter.

3.3.5 Logistic Regression

A logistic regression classifier is a discriminative model. This is in contrast to Naïve Bayes, which uses a generative approach. Instead, an logistic regression model will learn the boundary between classes, rather than how the data was distributed. Likewise to Naïve Bayes, it has been found to be effective on NLP problems [17].

Also known as a maximum entropy classifier (MaxEnt), an logistic regression model will try to find the probability of a specific outcome based on the relationship between all the features. Doing so by directly estimating $P(y = k|x)$, where k represents the class and x the feature vector. Rather than the joint distribution, which was the case with Naïve Bayes.

Logistic regression is similar to that of linear regression. However, instead of fitting a line in order to predict a continuous value, such as size. The logistic regression will predict if something is true or false, which it does by fitting a logistic (sigmoid) function. This function, which is an "S" shaped curve, can be expressed as seen in Eq. 3.8.

$$P(y = 1|x) = P(x) = \frac{1}{1 + e^{-(a+bx)}} \tag{3.8}$$

The way the function fits data is by converting a linear function $P(x) = a + bx$ into a range from 0 to 1. Writing the function in a more admissible form makes it possible to express the function as seen in Eq. 3.9.

$$P(x) = \log\left(\frac{p(x)}{1 - p(x)}\right) = a + bx \tag{3.9}$$

From this function, probabilities can be calculated using the concept of odds ratio. This is a concept that is based on calculating the ratio of the odds of an event happening to not happening. The odds ratio, namely $p(x)/(1 - p(x))$, can be seen inside the logarithmic expression of the function. The final calculated estimate will not only make it possible to predict a true or false based on some scenario but, also make classifications of a document as either a TD- or non-TD discussion.

Likewise to the Naïve Bayes classifier, the implementation of an logistic regression model has been achieved using the scikit-learn library with Python. Furthermore, the same preprocessing and text vectorization steps that was previously explained in this chapter has also been used. In addition to this, TF-IDF have also been applied to the data. The results from using the classifier on both the manually labeled dataset, as well as the data from the projects, can be seen in the results section.

3.3.6 Recurrent neural network

A recurrent neural network (RNN) is the third text classification method that has been tested in this thesis. The RNN will together with the dataset and word embeddings use deep learning to make predictions, which it does through sentiment analysis. RNNs have been found to have great results on sequential data, which is the case for textual data since the text is naturally sequential. However, RNNs have also been found to not generalize very well on limited training data [31]. In any case, using the dataset along with the word embeddings may help the network make correct predictions.

Word embeddings are models for word representations [40]. These models will be able to represent words from a vocabulary as vectors of real numbers. I.e., a string such as "hello" may look something like [0.1384, 0.3775, 1.429]. The way word vectors are decided is by calculating semantics as a mathematical distance. This makes it so that words similar to each other have a short distance, and vice versa. For example, the words "dog" and "puppy" will be mathematically closer than "train" and "bread".

Another important aspect of word embeddings is how a model's input matrices are treated. In spite of how many unique words there are in a text corpus, the columns of the input matrices will remain static. I.e., instead of defining single words in a text, the relationship between each of the words is defined. The relationship between the words can therefore be mapped as distance (e.g., 3D space of real numbers [X, Y, Z]). Rather than individually representing words in a matrix, where the columns are the words (also known as one-hot encoding).

For example, when applying one-hot encoded vectors for a sentence like “this needs to be refactored”' with 5 unique words. Vectors will be generated in $\mathbb{R}^5$ and each word indicate a state in the vectors. I.e., “this” transforms into [1, 0, 0, 0, 0], “needs” to [0, 1, 0, 0, 0], “to” to [0, 0, 1, 0, 0] and so forth. If the same logic were to be applied to n unique words, it would create vectors in $\mathbb{R}^n$ and grow exponentially. Thus, create space and computation problems. However, if instead word embeddings were to be used, it would fit the text into a fixed dimension. For example, a 3-dimensional of the same sentence could make “this” transform into [0.15, 0.21, 0.58], “needs” to [0.24, 0.28, 0.62], “to” to [0.18, 0.26, 0.67] and so forth.

For this thesis, word embeddings from the GloVe open source project have been used [14]. Specifically, word embeddings with a dimension of 300 and a vocabulary of 2.2 million words. GloVe is an open source project from Stanford University that uses a Euclidean distance to measure the semantic or linguistic similarity between words [41]. The position of the vectors of real numbers is then not only learned from the text itself but also from the words that surround it. Implementing and using the word embeddings and RNN was completed with Python and the Keras library [15].

Likewise to the ML models, the text used for the RNN is preprocessed in the same way. I.e., the same data cleaning steps is applied to the dataset that will be used to train the network. It is also from this dataset that a word embedding matrix is created together with the word embeddings, which will make the input for the network. In this case, that input will be a tensor input. The tensor input, which is a mathematical object that holds data in n dimensions, is used to train a deep learning model. Then, subsequently, make predictions on unseen OSS project data. The data from the projects have also had the same data cleaning steps as the dataset.

The reason behind using an RNN to perform NLP tasks, compared to a simple artificial neural network (ANN), is that RNNs are recurrent and hence fit for sequence modeling. Sequences are an important part of language, considering that the sequence of words helps to define their meaning. For example, the sentence “how are you?” is different from “are you how?”. Even though they have the same words in them, the latter does not make sense. The reason for this is because of the sequence of the words.

In brief, a neural network is built on layers that are connected to each other, much like a circuit that connects nodes. However, instead of a straightforward logic like the ones found in circuitries, the nodes in a neural network try to mimick brain neurons. A simple neural network can be summarized with three separate layers: one input layer, a set of hidden layers, and

[14] https://nlp.stanford.edu/projects/glove/
[15] https://keras.io/

an output layer. The input layer takes an input and passes the data to the hidden layers. It is in this layer that mathematical functions are applied to the data and where computations happen, such as automatic feature creation, data transformation, and so forth. Lastly, the output layer is the layer that will end up storing the results. An illustration of a simple artificial neural network can be seen in Fig. 3.7.

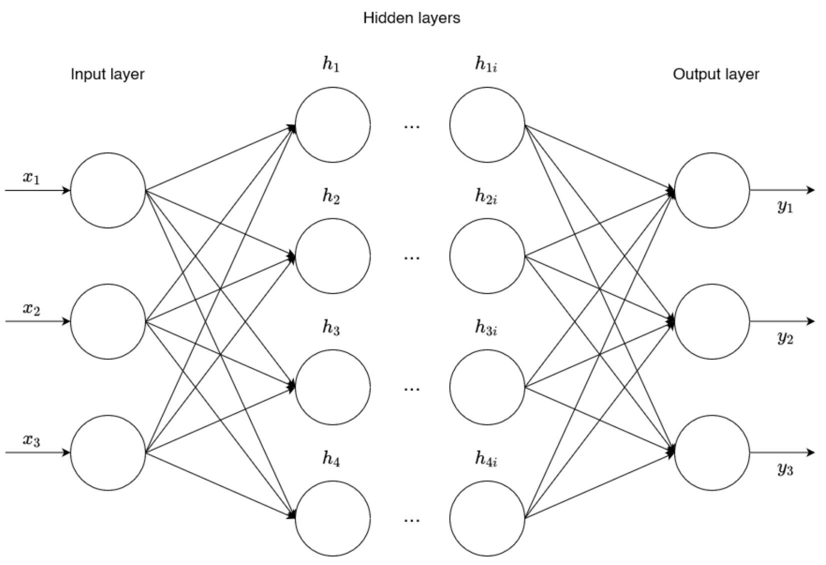


Figure 3.7: Simple artificial neural network

The artificial neurons (nodes) in the network are loosely based on biological neurons. They perform some processing based on input, and from that produce some form of output. Activating or "firing up" any of the neurons is done under a condition known as a threshold. This threshold is also what differentiates one neuron from the next. An example of a neuron firing up could be that a neuron takes three inputs x_1, x_2 and x_3 (just like in Fig. 3.7), for which it will produce some output y. In order to simplify the example, there will only be a single neuron h_1 in the hidden layer where this neuron has a function $f(x)$. This function is to add the two inputs together using addition if the two are equal to or greater than 10. In this case, the threshold can be summarized as seen in Eq. 3.10, where the threshold would have been equal to 10 and the bias is the negative threshold. It can be noted that all neurons in a layer share the same bias.

$$x_1 + x_2 + x_3 - \text{threshold} > 0 \quad \wedge \quad x_1 + x_2 + x_3 + \text{bias} > 0 \tag{3.10}$$

Furthermore, another important thing with neurons is that they can have weights. More specifically, the ability to add importance to certain input. This makes it so that certain inputs can have different weights depending on different parameters, where the weights will be able to represent the connection between the neurons. Making it so that one neuron can have a much greater influence compared to another. Finally, there are activation functions. This will in short dictate how computation inside the ANN is completed, where the activation function is responsible for computing the biases and the weighted sum from inputs.

RNNs on the other hand, only differ from ANN in that they are recurrent. While a traditional ANN assumes that each input and output will be independent of one another, an RNN will base its computation on a sequence's prior elements. An illustration of an RNN can be seen in Fig. 3.8.

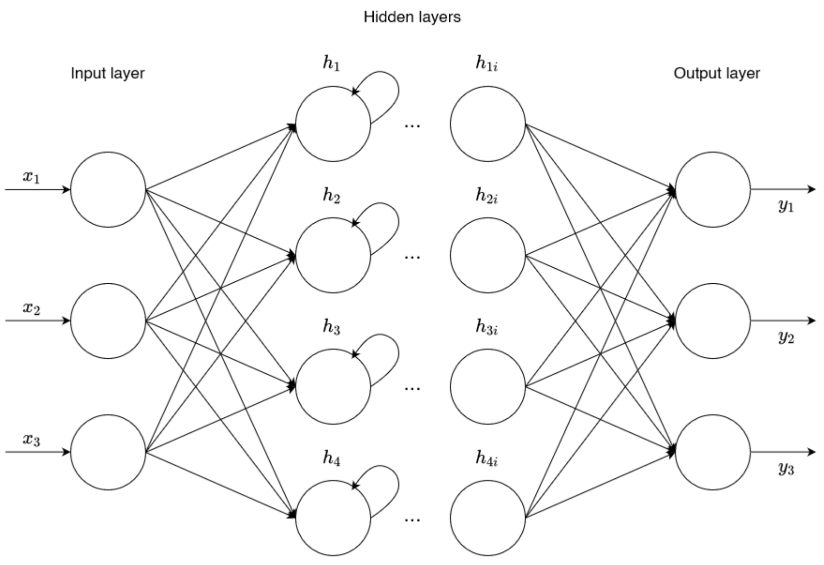


Figure 3.8: Simple recurrent neural network

The recurrent property makes it possible to temporarily store parts of a series of sequential data so that the respective locations of the different parts are considered. I.e., this makes it so that the relative words in the developer discussions will be weighted based on their importance, which is then in relation to the other words in the discussion.

As a result of this, an RNN is convenient for classifying text through sentiment analysis. As it will not only be able to pick up any nuances in the text that other models may not. Considering that the order in which words appear is taken into consideration, but also be able to store words based on their relative position and later use that in computations.

3.4 Measuring TD and DevOps metrics

This section presents how data points from the OSS projects have been used to calculate the DevOps metrics, as well as how their TD has been measured. Like it was described earlier in this thesis, the metrics are the velocity metrics as defined by DORA. The measured TD, on the other hand, will be the quantification of TD issues that are classified for each project.

Further, the size of the issues used to calculate the DevOps metrics and measure the TD has also been calculated. The reason for this is to include the weight of an issue, so that it may lead to more precise measures that will be proxies for the concepts. These proxies will then be defined by the operationalization process used for the measurements, where the proxies will result in variables that serve in place for the concepts, thus also resulting in a better construct validity for the thesis.

After the DevOps metrics have been calculated, the measures from it can be used in comparison with the measured TD issues. In this case, the metrics are correlated with open TD issues on a monthly basis. This may then give insight into how TD is prioritized and fixed, as well as how open TD issues correlate with the performance of software development, which is in this case velocity DevOps metrics. The results from both the calculations and the correlations are presented in the results chapter of this thesis.

3.4.1 Calculating issue size

Each of the issues in the OSS projects will have its own unique size, which may then indicate the effort needed to resolve them and how much they affect the projects. Some issues might be large, which will consequently require a lot of effort and affect the project on a large scale. Some might have a reasonable size, which may then take some effort and affect the project moderately. Others may be small, thus require less effort, and have a smaller influence. In any case, it is reasonable to assume that taking the size of the issues into consideration, will more accurately assess the measures used in combination with the issues. This factor will then be used with the DevOps metrics, in order to give more precise measures. In addition to the TD, where the size will give a TD issue its own weight.

For example, an issue that has taken a lot of effort to fix might have a bigger size than issues that were fixed more easily. Likewise, issues that are of greater size may in general have had a bigger impact on the project than those that were smaller. As compared to just using the number of issues, it will be possible to factor in effort and impact more easily with size-weighted issues. One could then, for example, differentiate between a big refactoring and a smaller update pushed to a project. As the big refactoring may have required a lot of code to be both written and deleted, whilst the smaller update may have required a much less significant change.

Calculating the size of an issue is done by calculating all the individual PR sizes s in a project, seen in Eq. 3.11. In this case, the size s is the amount of added and deleted lines in the PR. Furthermore, since the issues to PRs is a one-to-many relationship, it will be necessary to summarize all these s sizes together. An overall size $\bar{S}$ of an issue, as seen in Eq. 3.12, is therefore the product of all the PRs "additions" and "deletions" taken from each PR request. Also, as seen from Eq. 3.11, an "deletion" is only considered as half the effort of an "addition". The reason for this is that it requires less effort to make a line deletion, compared to adding new lines.

$$s = \text{additions} + \frac{\text{deletions}}{2} \tag{3.11}$$

$$\bar{S} = \sum_{i=1}^{n} s_i \tag{3.12}$$

An activity diagram of this process can be seen in Fig. 3.9. From the diagram, it can be seen that calculating the size for an issue, means that all the PR sizes for that one issue have to be combined together. This is the one-to-many relationship, as an issue may have none, one, or multiple PRs associated with it. If the issue has no PRs associated with it, then the size of the issue is none. Further, if there is only one PR associated with the issue, then the size of the issue is that PR, as can be seen in Eq. 3.11. Lastly, if the issue has multiple PRs associated with it, then the size is the total summarization of all the PR sizes, as seen in Eq. 3.11.

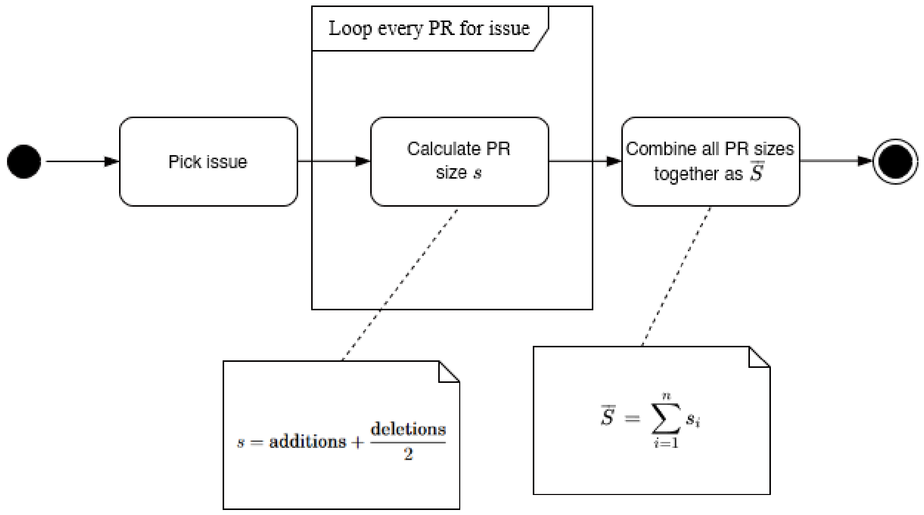

Loop every PR for issue
Pick issue
Calculate PR size s
Combine all PR sizes together as $\bar{S}$
$s = \text{additions} + \frac{\text{deletions}}{2}$
$\bar{S} = \sum_{i=1}^{n} s_i$

Figure 3.9: Process for calculating the individual issue sizes based on PRs.

An example of the calculation of issues can be seen in Table 3.5. In this table, there are four separate issues from the Beam project, namely: BEAM-10004, BEAM-10005, BEAM-10007, and BEAM-10009, as well as six individual and different PRs associated with these issues. All issues except from BEAM-10009 have a single PR associated with them. This means that their overall size $\bar{S}$ is simply s. However, for BEAM-10009 all the associated PR sizes s are added together in order to calculate the $\bar{S}$ for this issue.

Issue	...	**PRs**	**Additions**	**Deletions**	**PR size**	**Sum size**	...
BEAM-10004	...	12142	6	4	8.0	8.0	...
BEAM-10005	...	11855	139	2	140.0	140.0	...
BEAM-10007	...	11744	60	25	72.5	72.5	...
BEAM-10009	...	12764	478	35	495.5	848.0	...
BEAM-10009	...	12921	7	1	7.5	848.0	...
BEAM-10009	...	12573	339	12	345.0	848.0	...
...	...	...	...	...	...	...	...

Table 3.5: An example of calculating issue size. The "sum size" column has summarized the $\bar{S}$ size for the issues found in the "issue" column.

The data for each variable, namely the variables "deletions" and "additions", are fetched from the Github API in the same way as previously collected data has been described. Initially, it was possible to collect a third variable "files changed" from the API too. However, this was found to have a strong monotonic relationship with the other variables. The conclusion was therefore that this was a proxy variable for "additions" and "deletions", which could be left out of the size calculation. I.e., the variable is not relevant because it has a close relationship with the others. A correlation analysis for this can be seen in Fig. 3.10. In it the Spearman's rank correlation coefficient ρ for the "files changed" variable across all the projects is given. The p-value for each ranged from $1.628E-55$ to so low that it was indistinguishable from zero in Python.

The reason why the added and deleted lines in the PRs are used to represent size is simply because of how Git, and therefore also subsequently Github, seem to work. At the time of writing this thesis, it was not possible to use Github's API in order to get a pure LOC measure for each PR. However, Git describes the variables under the "git log" documentation as "Output only the last line of the –stat format containing a total number of modified files, as well as a number of added and deleted lines" [16]. In short, this means that if a PR has introduced 10 new lines and removed 15. Then the final size of that PR will be a total of 25 lines changed. This also applies to the modifications done to a line, where it will automatically be interpreted by Git as either an "addition" or "deletion".

[16]https://git-scm.com/docs/git-log

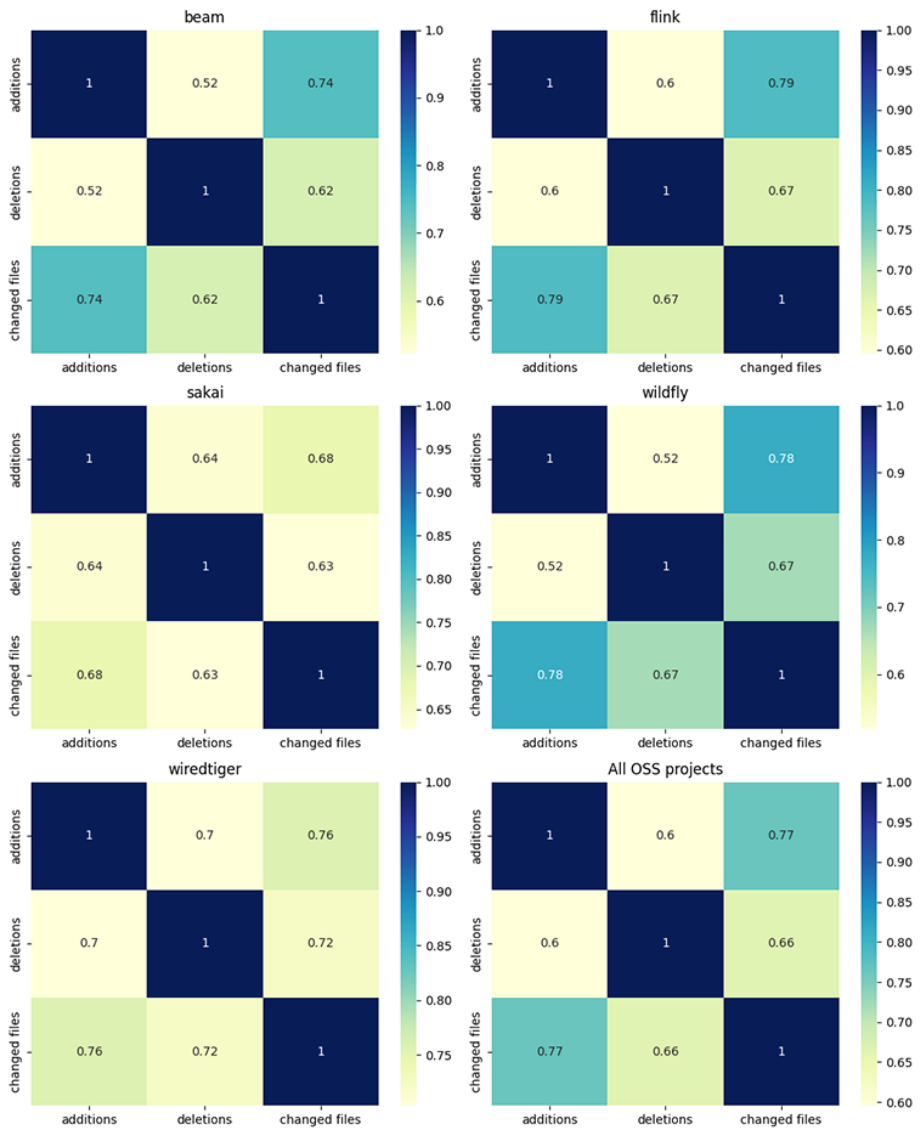


Figure 3.10: Correlation for the "files changed" variable

3.4.2 Lead time for changes

Lead time for changes is described by DORA as “the amount of time it takes a commit to get into production”. Further specifying that “for the primary application or service you work on, what is your lead time for changes (i.e., how long does it take to go from code commited to code successfully running in production)?”. In relation to calculating the metric, DORA continues by saying that calculating the lead time for changes is “done by using triggers with an SHA mapping back to the commits” [17].

In this case, the “amount of time it takes a commit to get into production” for each OSS project, is being interpreted as the time it takes for a PR to go from being created to merged. Other studies have, for example, seen the lead time for changes metric as simply the time it takes for an issue to go from open to being resolved [28]. This is different from that, as it is the commits in a PR, hence the code itself, that are being tracked, instead of tracking the issues directly.

As the data from this thesis comes from OSS projects, the repository commits and branches may in general differ compared to projects that aren’t OSS. Considering that practices for committing code may be different in closed projects. For OSS projects, however, as is the case with the projects in this thesis, there will be a system for PR requests where repositories are forked and PR requests made. When a PR is verified by any potential tests and a reviewer in a QA, the PR can then be merged into the project.

This makes it so that the lead time for changes has to be calculated by tracking the PRs instead of commits directly. A PR will then be able to represent one or more commits as a collection. This is slightly different from what DORA has defined lead time for changes, as they suggest tracking the commits individually. However, as this thesis associates issues to their PRs, it is necessary to map the PRs and track them over some given time, rather than the individual commits in a repository as DORA suggested.

In this case, the SHA mapping (also known as a hash) is a unique 40-character long identifier that is given to each commit. This identifier can be explained simply as a string of random characters generated by a cryptographic hash function. The string or SHA, makes it possible to differentiate one commit from others. Hence, why it is possible to map the commits by their SHA and then reliably use the data from the mapping.

However, as just briefly mentioned, it is the PRs that are being mapped in this thesis. This isn’t necessarily as definite as DORA has defined their mapping to be. As there will be a possibility that some commits, without having been linked to a PR beforehand, are created and pushed onto the different branches of a project. However, for the OSS projects, it seems like the observation is that pushing anything to the different branches without a PR is only an exception.

[17]https://cloud.google.com/blog/products/devops-sre/using-the-four-keys-to-measure-your-devops-performance

Calculating the lead time for changes for an issue can therefore be achieved by calculating the lead times for all PRs associated with that issue. Then, all of the lead times can be summed up as the overall lead time for a change for that specific issue. Just like how the size for each issue was calculated for all the issues. Further, in order to get a more precise measure, the metric has also been converted into lead time per change.

This will then take into account the size of the issues, thus increasing the construct validity. In contrast to just summarizing the number of lead times to measure the lead time for changes, the lead time per change will be able to factor in that it may take longer to make changes to larger issues, and vice versa. Simply summarizing the lead times would not have taken this into account and therefore resulted in a less reliable measure.

This conversion is achieved by dividing by the issue size, Eq. 3.13 describes this. Here, the lead time per change t is calculated by taking the sum of lead time for all PRs associated with an issue, then dividing that by the size $\bar{S}$ of the PRs. The distance function $d(x_i, y_i)$ describes the time distance between the creation and merging of the PR. This takes two dates, namely the creation- and merging date, and converts those into seconds. Then, the difference $|y - x|$ between the two is taken, which ultimately ends up being the lead time. Finally, the result is returned as lead time per change measured in seconds. This measure can then be interpreted as the number of seconds it took to make a change.

$$t = \sum_{i=1}^{n} \frac{d(x_i, y_i)}{\bar{S}_i} \tag{3.13}$$

A specific example of this could be to take any specific issue out from a project, for example, BEAM-10001 from the Beam project. Then, check how many PRs are associated with the issue, in this case it's just 1. When inspecting the PR, the creation date is Friday May 15, 2020 at 13:39 UTC, and merge date Friday May 15, 2020 at 20:32 UTC. As described by the distance function, these two dates has to be converted into seconds and then the difference between those two has to be taken. For these two dates the difference is 24,780 seconds. The difference is then divided by the size of the PR (which happens to be 5.5) as $\frac{24780}{5.5}$, which is equal to roughly 4,505 seconds. This means it took about 4,505 seconds to make the change.

Further, when using the lead time per changes measure in comparison with open TD issues, the projects were split up on a monthly basis. This made it possible to correlate the values as they changed over time with the number of open TD issues. Furthermore, both the first five and last five months of a project are removed for a more realistic representation. The reason for removing the first five is to exclude months that aren't likely affected by TD. The last five are removed because how the last few months might have issues that are still open. Keeping these months may therefore have affected the estimation negatively.

The overall lead time per change for a month will be the median of all lead time per change for the issues within that month. This will then return the center value for each of the months in the OSS projects, so that any outliers and skewed data will be accounted for and thus not affect the central tendency. When splitting up projects on a monthly basis for the lead time per change, the months were established by the creation date of all the issues. Fig. 3.11 tries to illustrate this relationship.

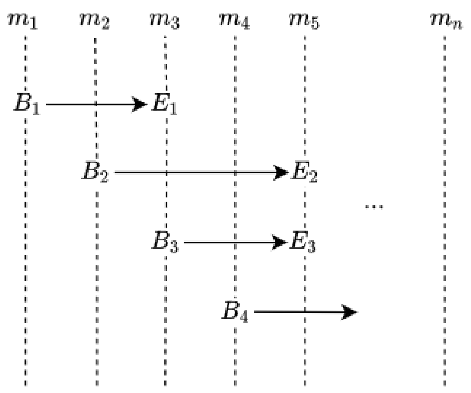


Figure 3.11: Illustration of monthly distribution m_i. Issue beginnings B_i will start at the beginning of a month, but issue endings E_i (resolved date) doesn't have to end at the same month.

From the figure, it can be seen that each month m_i has a fixed beginning B_i for the issues that fit within that month (from the 1st of the month till the last day of that month). However, the resolved date of the issues does not have to end E_i in the same month. I.e., all the issues that are created within one month, wouldn't necessarily have to be resolved within that month or the next. The reason for this is to look at how many changes it took to implement a solution for the TD issues, which is in this case achieved by looking at the data in retrospect. A specific example of this can be seen in Table 3.6.

	Issue	...	**Created**	**Resolved**	...
1	BEAM-502	...	2016-08-01 18:36:46+00:00	2016-08-03 19:10:44+00:00	...
2	BEAM-522	...	2016-08-03 21:04:14+00:00	2016-08-15 19:20:48+00:00	...
3	BEAM-523	...	2016-08-03 23:43:58+00:00	2016-08-11 18:18:02+00:00	...
4	BEAM-525	...	2016-08-04 00:42:48+00:00	2017-02-07 02:17:25+00:00	...
5	BEAM-528	...	2016-08-04 01:32:49+00:00	2016-10-05 19:51:34+00:00	...
6	BEAM-532	...	2016-08-04 01:49:13+00:00	2016-08-15 04:01:53+00:00	...
7	...	...	...	...	...

Table 3.6: Snippet of columns from a monthly distribution.

From the table, it can be seen that all the issues that have been included are created within a fixed month. In this case, that is August 2016. However, not all the issues are resolved within the same month. For example, rows 4 and 5 are all resolved in a different month. In fact, for row 4 the resolved date for this issue is in another year completely.

3.4.3 Deployment frequency

DORA has defined deployment frequency as "how often an organization successfully releases to production". They then go on to say "for the primary application or service you work on, how often does your organization deploy code to production or release it to end users?" [18]. I.e., the deployment frequency measures how rapidly deployments (updates, patches, features, etc.) is pushed to the end-users.

When calculating the deployment frequency, DORA states that "deployment frequency is the easiest metric to collect, because it only needs one table". This can then simply be a table that summarizes the deployments in a project. For this thesis, that will be the PRs (deployments) that have been merged into the repositories for the different OSS projects. They further specify that "it would be simple and straightforward to show daily deployment volume or to grab the average number of deployments per week, but the metric is deployment frequency, not volume" [19].

The deployment frequency is in this case measured over a monthly basis. Likewise to lead time for changes, the first and last five months from each project have been removed. This has been done to achieve a more realistic representation. The deployments over that monthly basis, namely the frequency, are calculated as the deployed value. This will take the issue size into account, rather than treating every deployment equally. Thus making it possible to differentiate between smaller and bigger deployments. As compared to just counting the number of deployments, the deployed value will then more accurately measure the deployment frequency.

For instance, treating a small and big issue equally for a given month will simply equal two deployments. However, separating them as the deployed value will instead be able to give a summarization of the value they add. In this case, that deployed value will be the size-weighted issues that are calculated based on their PRs. This would then make it possible to differentiate between, for example, a month that has two large deployments and a month that has one large and one small deployment.

Furthermore, when it comes to what constitutes a successful release, then this has been selected as the deployments that are resolved on a monthly basis. I.e., the deployments for a month will only include deployments marked as resolved within that month. This will exclude issues that

[18]https://cloud.google.com/blog/products/devops-sre/announcing-dora-2021-accelerate-state-of-devops-report
[19]https://cloud.google.com/blog/products/devops-sre/using-the-four-keys-to-measure-your-devops-performance

haven't been resolved for that month, which is in contrast to the lead time for a change, as it was calculated as the number of creations per month (regardless of the resolved date). The deployment frequency will instead be the number of resolves (successful deployments) per month.

The reason for selecting resolved issues as the successful deployments, rather than other measures such as issue labels (e.g., "fixed" or "closed"). Is that marking an issue as resolved is the final action that can be applied to an issue, regardless of the resolution. Although this wouldn't necessarily mean that each resolved issue has deployed something directly, such as code for a patch or feature. It can, however, be seen as an overview of the final resolution. This simplifies the process of what a deployment is and is not, which could otherwise be difficult to decide. The deployed value will also take this into account for the deployment frequency, as resolved issues that do not add value will not affect the deployment frequency for that month.

A specific example of this would be to say that there have been 20 successful deployments (resolved issues) for a month. These 20 issues are then summed up based on their issue sizes, just as the size of an issue was previously explained in this chapter. I.e., the final sum for the month is $\bar{S}_1 + \bar{S}_2 + ... + \bar{S}_{20}$, which could also be interpreted as the deployed value for that month. An example of this can be given with the snippet in Table 3.7.

	Key	...	**Created**	**Resolved**	...	**Sum size**
1	WT-2972	...	2016-10-13 07:01:20+00:00	2017-05-25 14:51:12+00:00	...	4938.0
2	WT-3158	...	2017-01-30 00:08:21+00:00	2017-05-12 01:22:38+00:00	..	920.5
3	WT-3248	...	2017-03-31 15:36:40+00:00	2017-05-26 20:41:17+00:00	...	33.0
4	WT-3258	...	2017-04-04 19:44:15+00:00	2017-05-26 02:30:35+00:00	...	366.5
5	WT-3264	...	2017-04-05 18:57:45+00:00	2017-05-22 16:47:40+00:00	...	241.5
6	WT-3303	...	2017-05-01 00:02:32+00:00	2017-05-19 05:07:20+00:00	...	53.0
7	..	...	...	..	...	..

Table 3.7: Snippet of columns from a monthly distribution.

First of all, as it can be seen from the table, the resolved date is fixed for a specific month while the creation date is not. The latter is especially noticeable with the first row, as the issue for this row has been created in 2016. Further, in order to calculate the deployment frequency for the snippet of this month, the deployed value has to be summarized, which means that the sum size $\bar{S}$ has to be added together for the issues. In this case, that would be $4938 + 920.5 + 33 + 366.5 + 241.5 + 53$ which is equal to 6552.5. The deployed value for that specific snippet is therefore 6552.5. The final unit of measurement from this deployed value will then be a product of the additions and deletions that were required to resolve the issues.

3.4.4 Measuring technical debt

Measuring the TD for each OSS project has been achieved by counting the TD issue classifications. More specifically, the number of issues classified as TD on a monthly basis. As a result of this, it is possible to compare the TD issues with the DevOps metrics in a correlation to open TD issues. Thus making it possible to gain insight into how TD is prioritized and fixed, as well as how the DevOps metrics are affected by open TD issues.

The monthly distribution of issues is decided by which of the two DevOps metrics is measured. However, the measured TD that is correlated with the metrics will be the number of TD issues that are opened on a monthly basis, regardless of the date they are resolved. As seen in the snippet from Table 3.8, issues are separated as either TD issues (signified by a numeric value of 1) or non-TD issues (signified by 0).

Further, as the DevOps metrics were weighted based on their issue size, TD items are being weighted too. In this case, the TD issues are multiplied by their issue size. Eq 3.14 describes the relationship between weighted TD issues and all the other issues. From the equation, it can be seen that an issue, which has either a numerical value of either 1 (TD issue) or 0 (non-TD issue), will be multiplied by its corresponding weight.

$$\sum_{i=1}^{n} a_i \bar{S}_i, \quad a \in \{0, 1\} \tag{3.14}$$

This makes it so that only issues marked as TD, namely given the numerical value 1, are included in the calculation. When splitting TD issues up on a monthly basis, this calculation will be the summarization of all the TD issues multiplied by their weight for that month. A specific example of this can be seen from the data in Table 3.8.

	key	**created**	**resolved**	**sum size**	**td**
1	BEAM-502	2016-08-01 18:36:46+00:00	2016-08-03 19:10:44+00:00	21.5	0
2	BEAM-522	2016-08-03 21:04:14+00:00	2016-08-15 19:20:48+00:00	42.0	1
3	BEAM-523	2016-08-03 23:43:58+00:00	2016-08-11 18:18:02+00:00	4.5	0
4	BEAM-525	2016-08-04 00:42:48+00:00	2017-02-07 02:17:25+00:00	1.5	0
5	BEAM-528	2016-08-04 01:32:49+00:00	2016-10-05 19:51:34+00:00	422.0	1
6	...	...	...	...	...

Table 3.8: Snippet of month with TD issues

In this case, the monthly summarization will be $0 \cdot 21.5 + 1 \cdot 42.0 + 0 \cdot 4.5 + 0 \cdot 1.5 + 1 \cdot 422.0$, which equals to 464. Following this sequence for the whole month, it is possible to use the final result in correlation with the DevOps metrics. Since the final result for all TD issues weighted per month will be able to represent the open TD issues, as well as their importance of them.

Chapter 4

Results

In chapter 4, the results from the methodology are presented. Each section in this chapter will first start out by briefly summarizing the processes related to the methodology, before moving on to analyzing and presenting the final results. The collected data section gives an overview of the data from the OSS projects after it has been extracted and processed. The results from the collected and processed data are used for both RQ1 and RQ2.

4.1 Collected data

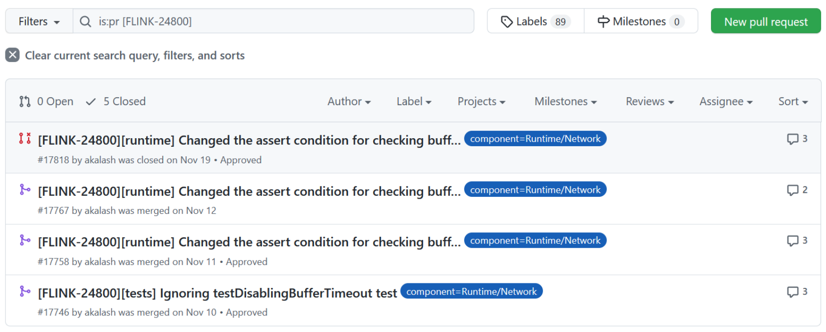
A summary of the extracted data from the OSS projects can be seen in Table 4.1. This summarizes the preliminary data that was collected from both Jira and Github. The data was further prepared and processed for the RQs so that it could be used with the classification models and DevOps metrics. The Jira issues field is the number of Jira issues that is unique, and that have at least one PR associated with them. The number of PRs is all the PRs that are associated with a specific Jira issue.

Name	**Jira issues**	**Pull-requests**	**Merges**	**Resolves**	**Comments**
Beam	6,925	11,249	8,073	6,117	22,583
Flink	13,000	15,267	4,684	12,669	113,182
Sakai	6,615	7,424	6,921	6,433	21,672
Wildfly	6,673	8,194	6,373	6,597	15,612
WiredTiger	3,794	4,917	4,472	3770	34,636
Sum	**37,007**	**47,051**	**30,523**	**35,586**	**207,685**

Table 4.1: Collected data

As seen from table 4.1, there is a wide variation between the projects. For example, the number of issues compared to PRs comments compared to issues and PRs, as well as the ratio between merged PRs and resolved Jira issues. This may be an indication of how each project may have different methodologies and ways of structuring its SDLC. For example, some projects, such as Beam, Sakai, and WiredTiger, have a greater amount of merged PRs compared to resolved Jira issues. While the other projects, namely Flink and Wildfly, have the exact opposite of that.

There can be several reasons for this. For example, one reason why there might be more merged PRs compared to resolved issues could be that an issue has taken several PRs in order to fully resolve. An example of this can be seen in Fig. 4.1, where an issue has been split up into three merged PRs and one closed PR. In this case, the PRs are not only merged into the Master branch [1] but also such as backports into previous releases. In short, this means that the PRs are not only merged into the present release but also into previous releases of the project.



[FLINK-24800][runtime] Changed the assert condition for checking buff... component=Runtime/Network 3
#17818 by akalash was closed on Nov 19 • Approved

[FLINK-24800][runtime] Changed the assert condition for checking buff... component=Runtime/Network 2
#17767 by akalash was merged on Nov 12

[FLINK-24800][runtime] Changed the assert condition for checking buff... component=Runtime/Network 3
#17758 by akalash was merged on Nov 11 • Approved

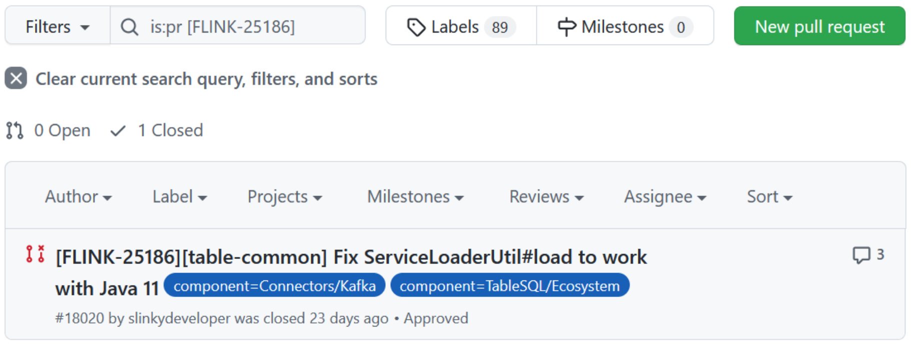
[FLINK-24800][tests] Ignoring testDisablingBufferTimeout test component=Runtime/Network 3
#17746 by akalash was merged on Nov 10 • Approved

Figure 4.1: Multiple PRs for a single Jira issue

Furthermore, a reason for why there might be more resolved Jira issues compared to merged PRs. Might potentially be due to the processes behind how issues are created, as well as how PRs are handled. One example of this could be that unnecessary issues would not have needed a PR in order to be resolved. Another example could be that the commits in a PR are not merged with the PR itself but, that the contributions in the PR are instead used in another commit for the codebase.

The latter can be observed in the PR seen in Fig. 4.2. This is a PR that is linked to a Jira issue that has been resolved with a resolution set as fixed. However, even though the Jira issue is resolved and the PR itself has been approved, its status of it is closed and thus the PR is not merged. I.e., the contributions from the PR have not been introduced into the codebase via merging that specific PR.

[1]https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-branches



[FLINK-25186][table-common] Fix ServiceLoaderUtil#load to work with Java 11 component=Connectors/Kafka component=TableSQL/Ecosystem

#18020 by slinkydeveloper was closed 23 days ago • Approved

Figure 4.2: Closed PR for a Jira issue that has been marked as fixed

Instead, the contributions from the PR were merged into the codebase through another separate commit. The reviewer had then decided to make a commit that was co-authored with the author of the PR, and then push that directly onto the Master branch. This commit would then include contributions from the original PR and resolve the issue. Furthermore, all things considered, it seems like the data that has been extracted has a large scope. In that, there are differences between the technical background of each project, the methodologies, and so forth.

The results from further processing of the data have been summarized in Table 4.2. This table is the result of the process of cleaning the OSS projects, which is done just as it was explained in the methodology chapter of this thesis. The data is based on unique Jira issues, which can have multiple PRs associated with it. E.g., For the Flink issue FLINK-24800 counts as one issue, even though FLINK-24800 can have multiple PRs. As seen in Fig. 4.1, the FLINK-24800 issue has four PRs that are associated with the issue. This data is that which has been used for RQ1 and RQ2.

	Beam	**Flink**	**Sakai**	**Wildfly**	**WiredTiger**
Total issues	5,786	10,729	6,465	5,353	3,452

Table 4.2: Final amount of issues from the OSS projects.

4.2 RQ1 quantifying how TD evolves from developer discussions

Prioritizing TD is one of the most important undertakings for managing TD in software projects. The process is used to help determine how resources in a project are allocated, including such as decision-making on new features or refactoring TD debt [26]. In order to perform a TD prioritization, relevant information regarding the TD must be made available.

For this thesis, RQ1 looked into how NLP can be applied to OSS projects as a method for quantifying TD. Specifically, how TD evolves out of developer discussions by generalizing NLP methods to a number of projects from the open source community. The purpose of this was to try to further refine the work carried out by Ozkaya et. al [37], which simply looked at a specific case with the Chromium project. Their study used ML and NLP to detect TD discussions from the project but fell short of creating an oracle for this type of approach. Thus, RQ1 has been directed towards trying to improve this. Not only did this include improving the accuracy for classifying TD issues, but also improving feature engineering and seeing if the classification models can be generalized for multiple projects.

To be more precise, the aim of RQ1 is therefore to see if developer discussions, that otherwise have design concerns that are hard to detect, can be generalized and marked as TD issues. That is, discussions that may express concerns related to more than what code inspection alone can uncover. For example, architectural design concerns, ad-hoc development choices, as well as other factors that may accumulate interest over time. As Ozkaya et. al [37] concluded in their study, this can be invaluable information for keeping track of TD in large projects, as well as making decisions related to TD prioritization. TD issues can then be related to their corresponding discussions, as they said, and potentially be especially useful for making decisions related to timely resolutions, decisions, and communication.

Furthermore, both the ML models and RNN for this thesis are mostly evaluated on their f1-score, specifically on the f1-score related to classifying TD issues rather than non-TD issues. The f1-score is a metric that better accentuates the performance of a classifier, compared to other metrics such as the accuracy score alone, which is simply the percentage of predictions the models got right. The reason for using this metric is that it combines both the precision and recall of the classifiers into a single score. Obtaining these scores are achieved by testing the classifiers.

The precision metric is the ratio between the true positives and all the positives in the classification. I.e., the precision can be measured as a percentage of correctly identified TD issues divided by all the TD issues. Recall, also known as sensitivity, is on the other hand slightly different. It measures the proportion of correctly identified true positives.

Testing the classification models has been done by splitting the dataset used for supervised learning, and then sharing the dataset for both training and testing. Both the ML models have been tested with a straightforward train/test split with a ratio of 80/20. 80% of the dataset has then been used for supervised learning, and the remaining 20% used for testing. The RNN has been tested in the same way but on a different premise. Instead of using an 80/20 split, the dataset has been split up using k-fold cross-validation, which is further explained in the RNN section of this chapter.

Furthermore, all the ML models have been optimized and tuned using hyperparameter optimization. This form of model optimization involves finding hyperparameters that are the most optimal for the model's learning process, the parameters will thus be something that affects how the model behaves. Finding the hyperparameters has been achieved using a grid search approach. In short, a grid search technique deals with trying different combinations, so that the most optimal hyperparameters can be found by calculating the performance for each of the combinations.

The classification model that was ultimately selected was the logistic regression model, which can be seen in Table 4.6. Compared to the other models, it was this one that got the best results. This model was further verified through manual inspection, where a sample of the predictions from the model was reviewed. In order to avoid bias, the task was completed by another master's student from the University of Oslo in informatics, where the person was given a hundred random issues and asked to manually classify them as either TD or non-TD issues. The results from this were then matched with the predictions from the logistic regression model.

The student was not told why the classifications had to be manually marked. The reason for withholding this information was to not introduce any bias, as explaining the purpose behind the classifications might have influenced the outcome. However, the student was given instructions based on how to identify TD discussions, which was the rubric from Bellomo et al. [8]. The outcome of the manual classification was that 72 of the 100 classifications matched the predictions of the logistic regression model. This match seems to correspond with the performance scores that are achieved with the logistic regression model, which can be seen in Table 4.6.

4.2.1 Using the initial dataset

The initial dataset used for the RNN and ML models, which was mostly based on the study by Ozkaya et al. [37], achieved good classification results. However, this was only when the dataset was upsampled, especially for the logistic regression model. The reason for upsampling the dataset, compared to leaving it as it was, is that the class of non-TD issues was overrepresented. Upsampling the dataset will make it so that both the TD and non-TD classes are balanced. That is to say, represent TD issues and non-TD issues as either the same or roughly the same quantity [38].

The technique used for upsampling the data was a back-translation method [14]. This involves taking a random set from the underrepresented class, namely the TD issues, and then translating them to another language and back to the original. In this case, that meant translating a random set of TD issues from the source language English to German, which would then become the target language. After that, the target language was translated back into back-translated TD issues. As a result of this, the upsampled dataset could be used for both training and testing the classifier. This process was achieved by writing a script that did the back-translation, where both Python and Google Translate's Googletrans library was used.

The results for the logistic regression classifier can be seen in Table 4.3. In total, the upsampled dataset had 2,234 issues, where 1,787 issues were used for training the classifier and 447 issues for testing. As it can be seen from the table, the overall f1-score is 0.85, the AUROC is 0.84, precision is 0.82, recall is 0.88 and accuracy is 0.85. Of particular note is the results from the classification of TD discussions. As it is of most interest to classify the TD discussions correctly, compared to classifying issues that are not TD. The f1-score for this is 0.85, the recall is 0.88 and the precision is slightly lower than for non-TD issues with 0.82.

	Precision	**Recall**	**f1-score**	**Support**
Non-TD discussions	0.88	0.81	0.84	225
TD discussions	0.82	0.88	0.85	222

Accuracy			0.85	447
Macro avg	0.85	0.85	0.85	447
Weighted avg	0.85	0.85	0.85	447

	Precision	**Recall**	**f1-score**	**AUROC**
Overall	0.82	0.88	0.85	0.84

Table 4.3: Logistic regression classifications with upsampled initial dataset

These results are slightly lower than the AUROC- and accuracy scores from Ozkaya et al. [37], which can be seen in Table 4.4. In total, they tested three separate models, but their main model achieved an AUROC of 0.88 and an accuracy of 0.87. However, the performance for precision and recall is significantly higher than their overall precision of 0.40 and recall of 0.62.

Performance metrics

	Accuracy	**Precision**	**Recall**	**AUROC**
No TD	0.90	NA	0.00	0.50
Keyphrase query	0.83	0.26	0.35	0.62
Main model	0.87	0.40	0.62	0.88

Table 4.4: Weighted performance metrics from Ozkaya et al. [37].

4.2.2 Improving the dataset

In spite of the good results from the initial dataset, which was based on the Chromium project and a few SATD issues marked by developers from the OSS project. The classification models did not seem to generalize appropriately on unseen data. This was the case for the data from the OSS projects, where classifications of TD could vary from as little as 11.96% to as much as 76.47% between the projects. The predictions from the classification when using the initial dataset can be seen in Table 4.5.

As a consequence of this, it was decided that the initial dataset had to be extended with sufficient data. Making it so that the classifiers could learn to generalize properly on unseen data. In consideration that the OSS projects are so varied in technical- and organizational variety, the decision was to extend the dataset with data from the OSS projects.

Predictions from the logistic regression model using initial dataset

Project	**Total issues**	**TD issues**	**non-TD issues**	**% of TD**
Beam	5,786	876	4,910	15.14%
Flink	10,729	8,204	2,525	76.47%
Sakai	6,465	773	5,692	11.96%
Wildfly	5,353	1727	3,626	32.28%
WiredTiger	3,452	2606	846	75.5%

Table 4.5: Predictions from the logistic regression classification model using the initial dataset.

As previously explained in the methodology chapter, the total amount of issues was increased from 1,639 to 2,189 for the new dataset. Additionally, the length of the text (characters) per issue was increased from 973 to 4,261 for TD issues. For non-TD issues, the number of characters increased from 1,185 to 7,540. Furthermore, although both the number of issues and length of text in the issues had increased with the new dataset, the TD issues class was still being underrepresented. As a result of this, the new dataset was balanced by undersampling the majority class, namely the TD issues. The method for undersampling the new dataset was to remove randomly selected TD issues. This would then result in a final dataset with a total of 1,501 issues, where 756 were TD issues and 745 non-TD issues.

The final dataset would then contain a mix of the initial dataset and new issues from the OSS projects. This dataset was then used for the Naïve Bayes and logistic regression ML classifiers. However, for the RNN both the upsampled initial dataset with the new issues were used. The predictions from the classifiers, for example, the logistic regression model, ranges from 10.3% - 17.06% issues being classified as TD, while the predictions from the RNN range from 20.46% - 31.23% TD. When it comes to the classification performance, the results from the Naïve Bayes model can be seen in Table 4.7, logistic regression model in Table 4.6, and RNN in 4.10.

The numbers from the logistic regression model are close to what Ozkaya et al. [37] predicted, which estimated 16.1% TD issues in the Chromium project. The RNN has on the other hand made predictions that are well above that. However, other studies have estimated that 2.4% - 31% of the files in a project contain SATD [42]. This makes the prediction results from the RNN seem reasonable, without going in-depth into them.

4.2.3 Machine learning models

As previously mentioned in the thesis, although the ML models using the initial dataset achieved good results when tested. The ML models would subsequently fail to generalize properly on unseen data if the initial dataset alone was used for training and testing. Thus, a new dataset had to replace the initial one, and then be used to train and test the classification models. From this, the classification report for the logistic regression classifier can be seen in Table 4.6. The report for the classifications made by the Naïve Bayes classifier can be seen in Table 4.7.

Logistic regression

	Precision	**Recall**	**f1-score**	**Support**
Non-TD discussions	0.73	0.76	0.74	146
TD discussions	0.76	0.73	0.75	155

Accuracy			0.74	301
Macro avg	0.74	0.74	0.74	301
Weighted avg	0.75	0.74	0.74	301

	Precision	**Recall**	**f1-score**	**AUROC**
Overall	0.76	0.73	0.74	0.74

Table 4.6: Logistic regression classifications with the new dataset.

Naïve Bayes

	Precision	**Recall**	**f1-score**	**Support**
Non-TD discussions	0.65	0.92	0.76	146
TD discussions	0.88	0.52	0.66	155

Accuracy			0.72	301
Macro avg	0.76	0.72	0.71	301
Weighted avg	0.77	0.72	0.71	301

	Precision	**Recall**	**f1-score**	**AUROC**
Overall	0.88	0.52	0.65	0.72

Table 4.7: Naïve Bayes classifications with the new dataset.

As seen from the tables, the logistic regression is able to make better classifications compared to the Naïve Bayes model. Although the overall f1-score for non-TD discussions aren't too different from each other, the logistic regression classifier has better f1-scores for both the TD discussions and overall. A diagram that compares the receiver operating characteristic (ROC) curves for both of the classifiers can be seen in Fig. 4.3.

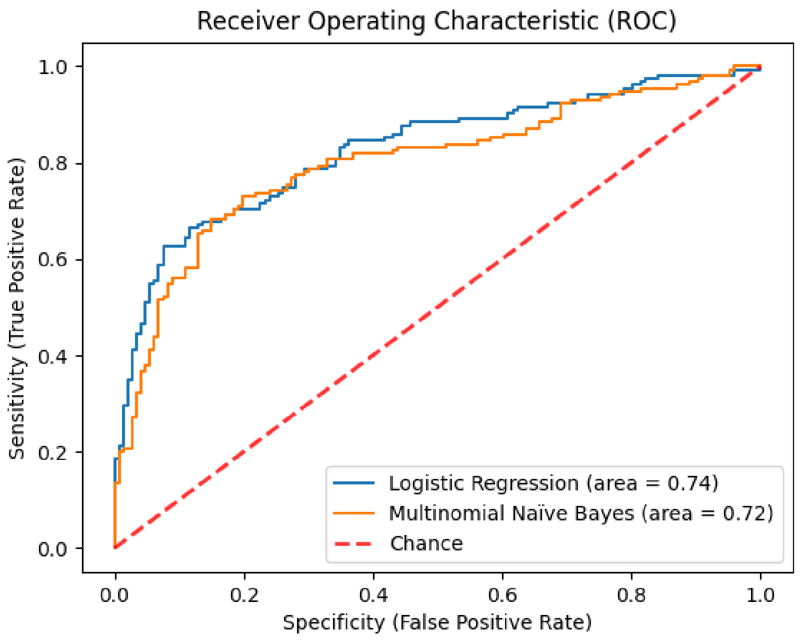


Figure 4.3: ROC curves for Naïve Bayes and logistic regression

The ROC curves describe the classification thresholds for the models, which are based on the two parameters true positive rates (TPR) and false-positive rate (FPR). The plots seen in the diagram are the relationship between TPR vs. FPR, which is shown as a curve for each of the different thresholds. In this case, the threshold for random chance is also included as a contrast to the two models. The threshold itself is the given probability. The TPR, also known as sensitivity or recall, describes the rate of positive classes being correctly predicted. I.e., correctly predicted TD discussions. FPR on the other hand describes the rate of positive classes being incorrectly predicted. I.e., an issue being predicted as TD when it isn't.

Both the models have had a classification threshold set as $P > 0.75$. Normally for the Python scikit-learn library, which has been used to implement the models, the threshold is set at 0.5. This means that a probability is converted to a binary value based on the threshold. For example, a probability of 0.55 or 0.95 would be converted to 1, which will classify it as TD. While a probability of 0.43 or 0.13 will be converted to 0, classifying it as non-TD. The reason for adjusting it to a higher threshold is that it is more important to classify fewer TD discussions correctly with high certainty, rather than the opposite. As there will be a big difference between a prediction of 0.95 and 0.55, thus thresholds are something that is dependent on the problem. It was therefore concluded that tuning the threshold from 0.5 to 0.75, would make more sense for the classifications.

Furthermore, because the logistic regression model outperformed the Naïve Bayes classifier, the logistic regression model has been chosen. The model made especially more correct predictions when taking the classification of TD into consideration. A Table 4.8 summarizes the predictions for the unseen OSS data using the logistic regression model.

Predictions from the logistic regression model

Project	Total issues	TD issues	non-TD issues	% of TD
Beam	5,786	596	5,190	10.3%
Flink	10,729	1,722	9,007	16.05%
Sakai	6,465	754	5,711	11.66%
Wildfly	5,353	639	4,715	11.94%
WiredTiger	3,452	589	2,863	17.06%

Table 4.8: Predictions from the logistic regression classification model.

The top 30 words with the highest TF-IDF score from the classification can be seen in Table 4.9. This is based on the new dataset during the training of the classifiers and is the same for both the Naïve Bayes- and logistic regression classifier. The weights are the TF-IDF score given to each word, and this represents the importance of that word.

#	Word	Weight	#	Word	Weight	#	Word	Weight
1	lastcompletedbuild	3.660	11	job	2.452	21	fix	1.946
2	beam	3.311	12	page	2.449	22	file	1.902
3	eviction	3.009	13	format	2.434	23	github	1.900
4	cursor	2.991	14	server	2.325	24	org	1.862
5	chrome	2.865	15	error	2.285	25	change	1.818
6	wiredtiger	2.763	16	branch	2.261	26	test	1.795
7	mongodb	2.753	17	java	2.177	27	the	1.775
8	memory	2.739	18	message	2.072	28	com	1.606
9	flink	2.697	19	build	2.055	29	issue	1.534
10	apache	2.553	20	add	1.959	30	http	1.381

Table 4.9: Top 30 words with highest TF-IDF score.

The word from the table are ranked after their importance. This makes it so that the most important word has the most weight, and vice versa. In this case, the most important word is “lastcompletedbuild” with a weight of 3.660. The least important word was “http” with a weight of 1.381. Furthermore, it can be noted that word “eviction” seems to be solely related to the Chromium project. The words “org”, “com” and “http” are likely part of URLs that have been split up because of the data pre-processing.

4.2.4 Recurrent neural network

Likewise to the ML models, the threshold for the RNN classifier has also been set to P $>$ 0.75. The results from the prediction using the RNN on the unseen data from the OSS projects can be seen in Table 4.10. It can be noted that k-fold cross-validation has been used as a statistical method to test the RNN. In this case, 10 folds (denoted as k=10) have been selected.

Recurrent neural network

	Accuracy	**Precision**	**Recall**	**f1-score**	**AUROC**
Overall (k=10)	0.725	0.726	0.723	0.721	0.723

Table 4.10: Classification report for the RNN classification model.

Cross-validation is used as a resampling procedure to estimate the performance of the RNN. By using the dataset with the k-fold cross-validation, the results are less biased compared to the ML models, which were just tested with a straightforward train and test split. The “k” refers to the number of so-called groups that the dataset is split into. In this case, that is 10-fold. For each group, the model will hold back one group and use the rest to train. After training on the data, the model will evaluate the predictions using the group that was held back as test data. The same process is then repeated ten times. The ROC curve for the classification, which takes the cross-validation into consideration, can be seen in Fig. 4.4.

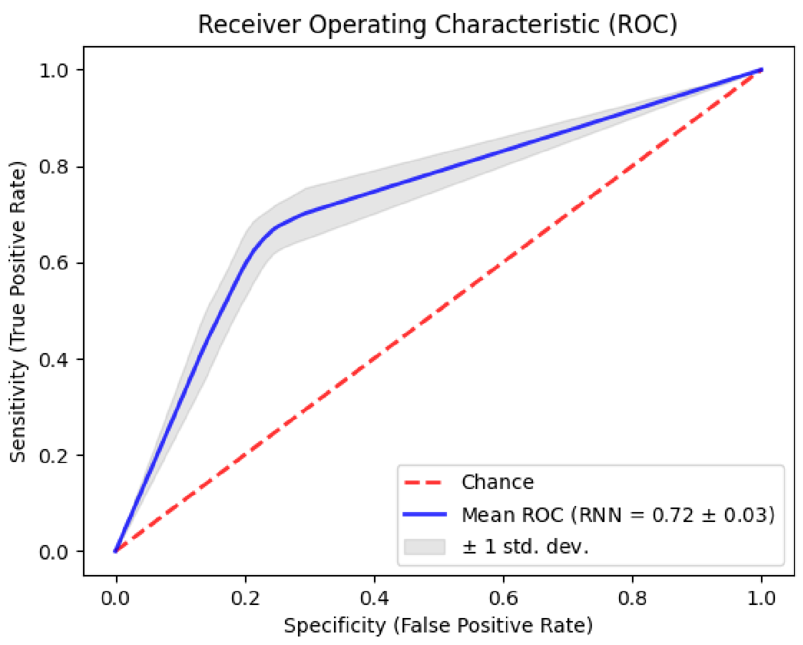


Figure 4.4: ROC curve for the RNN

When the RNN model is run on unseen data from the OSS projects, the most amount of TD predicted from the RNN is almost double that of the logistic regression model. A summary of the predictions can be seen in Table 4.11. From the table, it can be seen that the lowest amount of predicted TD is 20.46% and the highest is 31.23%. The highest is Beam, which is three times as much as the logistic regression model predicted with 10.3%.

Predictions from the RNN model

Project	Total issues	TD issues	non-TD issues	% of TD
Beam	5,786	1,807	3,979	31.23%
Flink	10,729	2,368	8,361	22.07%
Sakai	6,465	1,323	5,142	20.46%
Wildfly	5,353	1,188	4,165	22.19%
WiredTiger	3,452	851	2,601	24.65%

Table 4.11: Predictions from the RNN classification model.

The RNN's most important words and their respective weights can be seen in Table 4.12. Compared to the logistic regression model, the RNN seems to have identified completely different words. Some, such as "conflictingexternalidexception", "reuse" and "winxp" (possibly Windows XP), seem to be relatable TD references. While other words such as "winter", "translator", as well as numbers such as "459" seem to be more diffuse.

#	Word	Weight	#	Word	Weight
1	conflictingexternalidexception	2.638	16	relationship	2.083
2	winter	2.388	17	snuck	2.076
3	commit	2.388	18	compiler	2.055
4	implementing	2.290	19	rm	2.047
5	objectdefinproperty	2.272	20	translator	2.036
6	fa	2.271	21	clustering	2.029
7	blow	2.268	22	portalhanderexception	2.024
8	o	2.265	23	968	2.020
9	winxp	2.229	24	contain	2.020
10	reuse	2.217	25	committers	2.011
11	unstyled	2.196	26	addsqlfunctionmethod	2.011
12	seek	2.182	27	curtable	2.003
13	ported	2.155	28	drawer	2.003
14	owernship	2.113	29	lastcompletebuild	1.997
15	459	2.100	30	430886993	1.987

Table 4.12: Top 30 words from RNN model.

Furthermore, although the predictions are considerably different than that of the logistic regression model. The numbers are still within the range of what other studies have suggested. For example, Potdar et al. [42] estimated that between 2.4% - 31% of files in a project contained SATD. In addition to this, it can be pointed out that the predictions doesn't seem vary to much in range, with that the TD predictions are distributed similarly.

The RNN may, therefore, along with the word embeddings, have found different hidden patterns than what the other ML models have. Hence the reason for why some of the top words may be diffuse can simply be that they are related to code in the discussions. For example, the word “rm” may be a reference to the rm command from Unix-like operating systems, which is used as a command used to remove files on the operating system. This is something that can also be the case for other top words, such as “addsqlfunctionmethod”, which seem to be directly related to code.

If that happens to be true, then it may be logical that they are words related to TD. As removing files can be linked to activities like refactoring, and a word like “addsqlfunctionmethod” can possibly be related to concerns regarding database architecture. Considering that the word seems to mention SQL, which is a programming language for databases.

However, in spite of this and due to better scores from the logistic regression model. It was ultimately decided that this model and its predictions had to be chosen. Not only did it have a better overall f1-score than the RNN, but most importantly, a better f1-score for prediction TD issues. Further, the most important words for the logistic regression model also seem to make more sense. As the words such as “lastcompletedbuild”, “'fix’, “memory”, “change”, “error”, and so forth, seem to make more sense compared to many of the top words from the RNN.

4.3 RQ2 using quantified TD with DevOps metrics to obtain insight

This section summarizes the correlation analysis for the DevOps metrics. As previously mentioned in the methodology chapter, it is the two velocity DevOps metrics, namely lead time for changes and deployment frequency, that have been correlated with the quantified TD. In addition to this, both the quantified TD and DevOps metrics have also been correlated with the OSS projects’ Github repository size. This has been done because the results from the correlations might be influenced by the size of the project.

First, the tables for the correlations are presented. These will then include the correlations for the DevOps metrics correlated with the quantified TD, as well as the metrics and TD correlated with size. Then, the charts and plots for the respective correlations measured over time will be given.

4.3.1 Lead time for changes

The correlation between lead time for changes, which has for this thesis been further specified as lead time per changes, and the quantified TD can be seen in Table 4.13. As seen from this table, the Pearson’s correlation coefficient r, which simply measures linear correlation, doesn't seem to indicate any linear correlation between the lead time for changes and TD. Further indications of this can be seen when taking the p-values into consideration, as the results from these are not significant.

The p-values describe the statistical significance of the correlation. These are basically the probabilities that measure, given that the null hypothesis is true, the likelihood of the observations from the correlation being found. A small p-value of less than 0.05 (also known as the level of statistical significance) would generally mean that the results from the correlation will have significance. As a result, it will be able to clarify whether the results from the correlation are due to chance or not [2].

Furthermore, Spearman's rank correlation ρ, which has been highlighted in the tables, measures the monotonic relationship for the correlation. The monotonicity includes both the direction and strength of the variables. In other words, when there is a change in one variable, it would normally result in a specific change in the other variable. As seen from Table 4.13, there does seem to be a strong monotonic relationship between the two variables in projects Flink, Sakai, and WiredTiger, as well as a weak positive relationship for this with Wildfly. Although Beam seems to be a deviation from this, where the result is a weak negative relationship between the two variables. Based on the p-values, the results from Spearman's rank correlation also seem to be significant.

The Kendall's rank correlation coefficient, denoted by τ, seems to also result in notable correlations. This is a non-parametric measure that ranks the correlation between the variables on an ordinal scale, including both their direction and strength. Further, as it can be seen from Table 4.13, the p-values for τ, excluding Beam, seem to be significant. Although the results from the measure only range from very weak to moderate correlations, compared to ρ which had a higher correlation degree.

Lead time for changes correlated with technical debt

		Beam	**Flink**	**Sakai**	**Wildfly**	**WiredTiger**
Pearson	r	-0.030	0.038	0.000	0.026	0.104
	p-value	0.814	0.737	0.997	0.770	0.279
Spearman	ρ	**-0.259**	**0.685**	**0.690**	**0.245**	**0.602**
	p-value	**0.045**	**2.285e-12**	**1.034e-26**	**0.005**	**4.099e-12**
Kendall	τ	-0.172	0.486	0.512	0.164	0.460
	p-value	0.052	1.975e-10	3.891e-19	0.006	2.114e-10

Table 4.13: Correlation for lead time for changes.

4.3.2 Deployment frequency

When it comes to the deployment frequency metrics, which are measured as the deployed value. Then the results for this can be seen in Table 4.14. This table summarizes the correlation between the metric and TD, where unlike the lead time for changes, there does seem to exist a linear correlation between the OSS projects as seen with r. The r correlation degree ranges from a positive to very weak to strong correlation between the two. In the case of Flink, Sakai, and Wildfly the p-values also seem to be significant for the correlations for all the OSS projects.

Furthermore, both Kendall's τ coefficient and Spearman's ρ coefficient seem to not only give high correlations but also correlation results that are significant. This can be seen by their corresponding p-values in Table 4.14. Of particular note for the correlation analysis for deployment frequency is the ρ values for the OSS projects, which seem to range from moderate to very strong positive correlations between the projects.

Deployment frequency correlated with technical debt

		Beam	**Flink**	**Sakai**	**Wildfly**	**WiredTiger**
Pearson	r	0.154	0.467	0.450	0.610	0.119
	p-value	0.238	1.257e-05	2.511e-10	2.622e-14	0.217
Spearman	ρ	**0.543**	**0.729**	**0.917**	**0.716**	**0.661**
	p-value	**7.363e-06**	**1.688e-14**	**1.347e-72**	**2.867e-21**	**5.069e-15**
Kendall	τ	0.410	0.539	0.767	0.559	0.528
	p-value	4.155e-06	1.524e-12	1.628e-37	3.598e-19	1.171e-12

Table 4.14: Correlation for deployment frequency.

4.3.3 Project size

As briefly mentioned in the introduction of this section, both the DevOps metrics and the TD has also been correlated with the OSS project sizes. The reason for correlating the project size with DevOps metrics, as well as the TD, is to see if it has an influence on the measures. As both the quantified TD and metrics may be influenced by the project as it grows over time. For example, it may be that the larger the project gets, the more TD is accumulated. Likewise, the larger the project becomes, the harder it might be to deliver software frequently. This makes it an important factor to consider.

In this case, each project size is interpreted as the size of its corresponding Github repository as it grows over time. More specifically, as it grows over time on a monthly basis, measured in a total amount of line- deletions and additions. The latter is the same measure as explained in the methodology for measuring DevOps metrics. I.e., the sum total of the number of lines that have been added and deleted in the project.

Getting the repository size for each OSS project was achieved by writing a Python script with the Github API. This has been done in the same way as other forms of data from Github were collected, and described in the methodology chapter. The size variable was obtained by summarizing the number of line additions and deletions for each commits in a month. Further, as explained in the measuring DevOps metrics section, this total amount doesn't necessarily translate into LOC. Considering that Git interprets an addition or deletion as something that is purely based on a line addition or deletion, which also includes any modifications to a line.

The correlation for lead time with repository size can be seen in Table 4.15, deployment frequency correlated with repository size can be seen in Table 4.16, and TD correlated with repository size seen in 4.16. From the tables, it can be seen that the repository size seems to have some sort of impact on the TD and DevOps metrics. For example, for Sakai and lead time for changes correlated with size, where the ρ is as high as 0.690. The same applies to Flink's deployment frequency and TD correlated with size, where the results are ρ is 0.691 and 0.531 respectively.

However, contrary to the other correlations, it seems like there is a bigger variation in results when correlating with size. As when the quantified TD was simply correlated with the DevOps metrics, most of the OSS projects seemed to follow the same trend. However, this doesn't seem to be the case when size is considered for the DevOps metrics and quantified TD. The only exception to this seems to be the lead time for changes correlated with repository size, which can be seen in Table 4.15.

Lead time for changes correlated with repository size

		Beam	**Flink**	**Sakai**	**Wildfly**	**WiredTiger**
Pearson	r	-0.204	0.029	-0.025	-0.123	-0.086
	p-value	0.116	0.794	0.731	0.166	0.370
Spearman	ρ	**-0.212**	**0.300**	**-0.245**	**-0.364**	**-0.527**
	p-value	**0.103**	**0.006**	**0.000**	**2.486e-05**	**3.873e-09**
Kendall	τ	-0.151	0.204	-0.171	-0.248	-0.351
	p-value	0.087	0.007	0.001	3.668	1.230e-07

Table 4.15: Repository size correlated with lead time for changes over time.

Deployment frequency correlated with repository size

		Beam	**Flink**	**Sakai**	**Wildfly**	**WiredTiger**
Pearson	r	-0.025	0.358	0.051	0.268	0.118
	p-value	0.845	0.001	0.489	0.002	0.217
Spearman	ρ	**0.176**	**0.691**	**-0.042**	**0.089**	**-0.292**
	p-value	**0.176**	**1.211e-12**	**0.572**	**0.315**	**0.002**
Kendall	τ	0.115	0.493	-0.022	0.085	-0.156
	p-value	0.193	9.609e-11	0.679	0.1619	0.019

Table 4.16: Repository size correlated with deployment frequency over time.

Technical debt correlated with repository size

		Beam	**Flink**	**Sakai**	**Wildfly**	**WiredTiger**
Pearson	r	0.115	0.682	0.090	0.107	-0.030
	p-value	0.377	2.990e-12	0.226	0.227	0.751
Spearman	ρ	**0.244**	**0.531**	**-0.120**	**-0.115**	**-0.315**
	p-value	**0.059**	**4.010e-07**	**0.107**	**0.194**	**0.000**
Kendall	τ	0.166	0.384	-0.086	-0.080	-0.216
	p-value	0.061	4.547e-07	0.115	0.192	0.002

Table 4.17: Repository size correlated with technical debt over time.

4.3.4 Deployment frequency line charts

Line charts for the deployment frequency for each of the OSS projects can be seen in this section. The charts feature deployment frequency compared to the technical debt over time on a monthly basis. The monthly distribution can be seen from the dates on the x-axis that is formatted with a date order of YY/MM/DD. Both the deployment frequency and TD can be seen on the y-axes in the charts. The TD is on the left y-axis and deployment frequency is on the right. It should be noted that the y-axes may have different values.

For example, the line chart for Flink, seen in Fig. 4.6, has two y-axes with different values expressed in scientific notation (seen in the top corners), where the y-axis for deployment frequency is twice the size of the y-axis for weighted TD. The other OSS projects can be seen in Fig. 4.5 for Beam, Fig. 4.7 for Sakai, Fig. 4.8 for Wildfly and Fig. 4.9 for WiredTiger.

In the charts, the deployment frequency is the deployed value for a month over the project's lifespan. I.e., just as it was explained in the chapter for measuring DevOps metrics, the deployment frequency takes the issue sizes into account as the deployed value. The y-axis for open TD issues that are weighted, represents the number of TD issues that are opened per month and their size. Furthermore, certain OSS projects have earlier activity than what can be seen in some of the charts. The reason for excluding this from the charts (not the correlation analysis) was that this activity was minimal. Thus, excluding it made the charts more representable. A little arrow in the chart will indicate that the project had earlier activity.

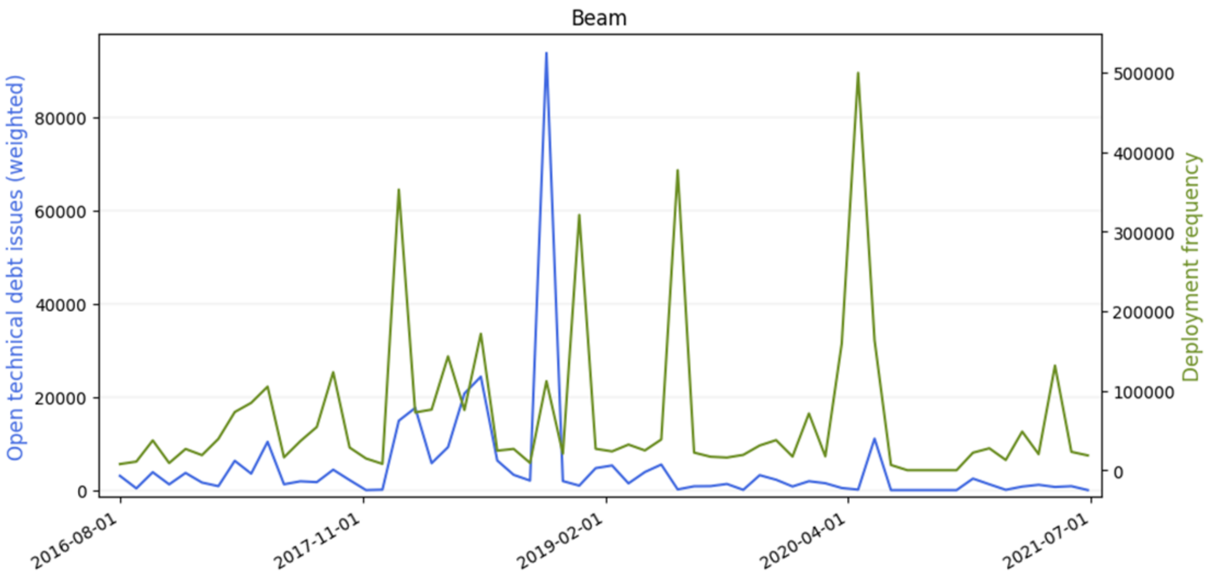


Figure 4.5: Line chart of deployment frequency and technical debt on a monthly basis for Beam. N.B., there are two different y-axes.

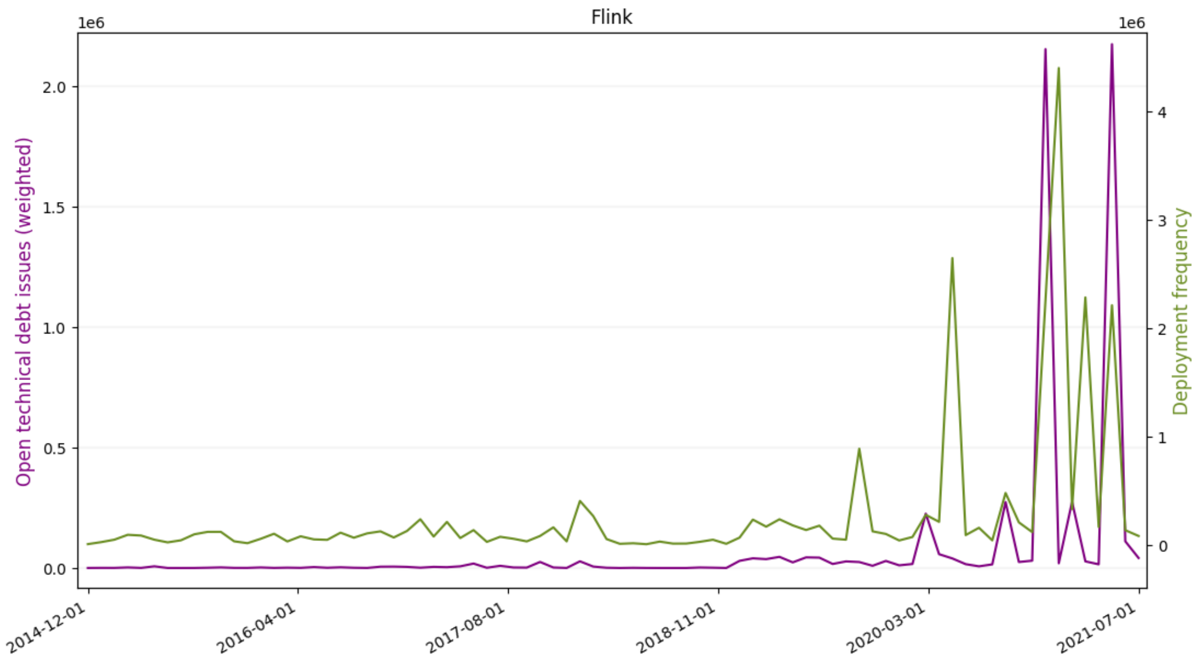


Figure 4.6: Line chart of deployment frequency and technical debt on a monthly basis for Flink. N.B., there are two different y-axes.

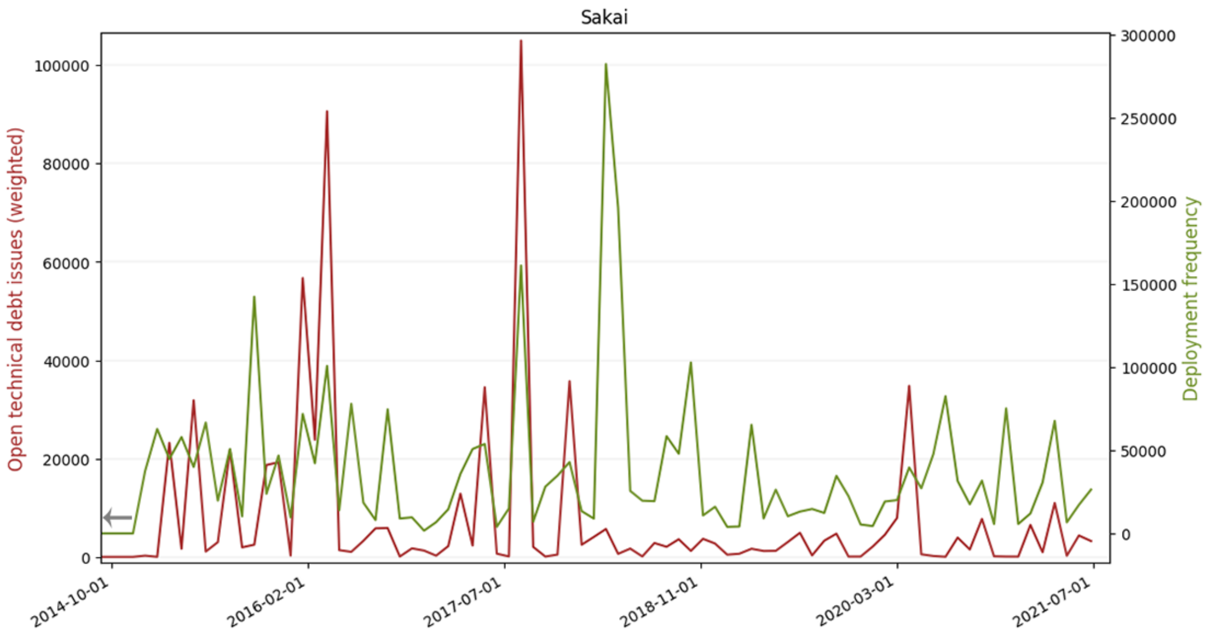


Figure 4.7: Line chart of deployment frequency and technical debt on a monthly basis for Sakai. N.B., there are two different y-axes.

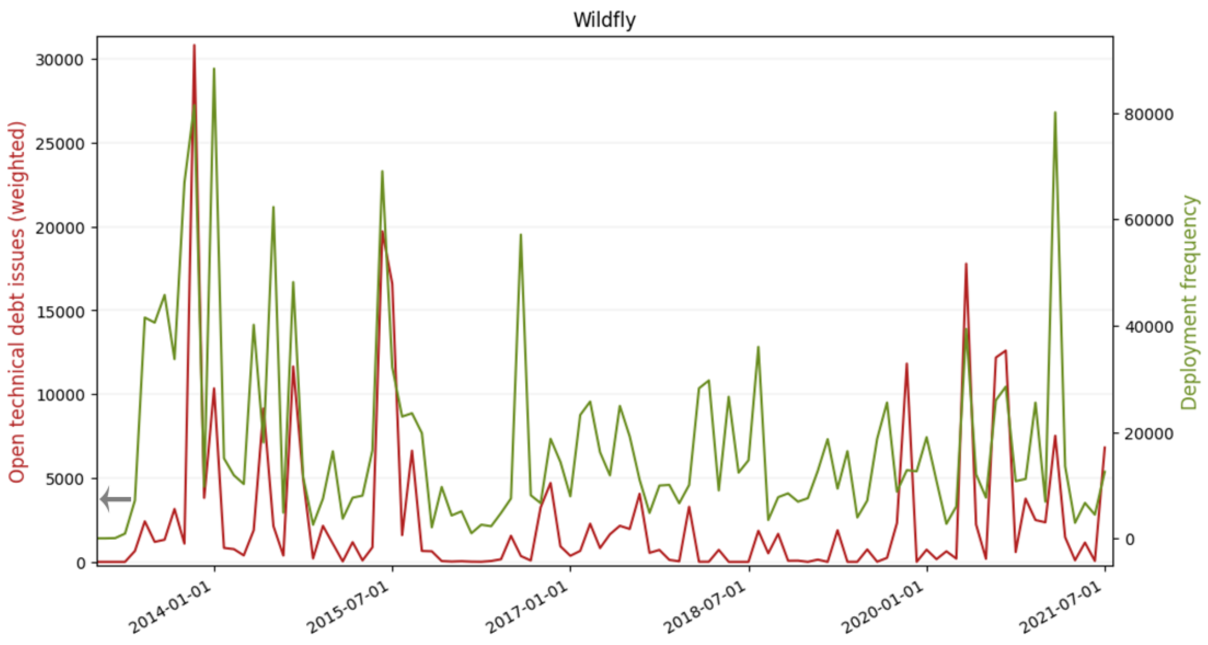


Figure 4.8: Line chart of deployment frequency and technical debt on a monthly basis for Wildfly. N.B., there are two different y-axes.

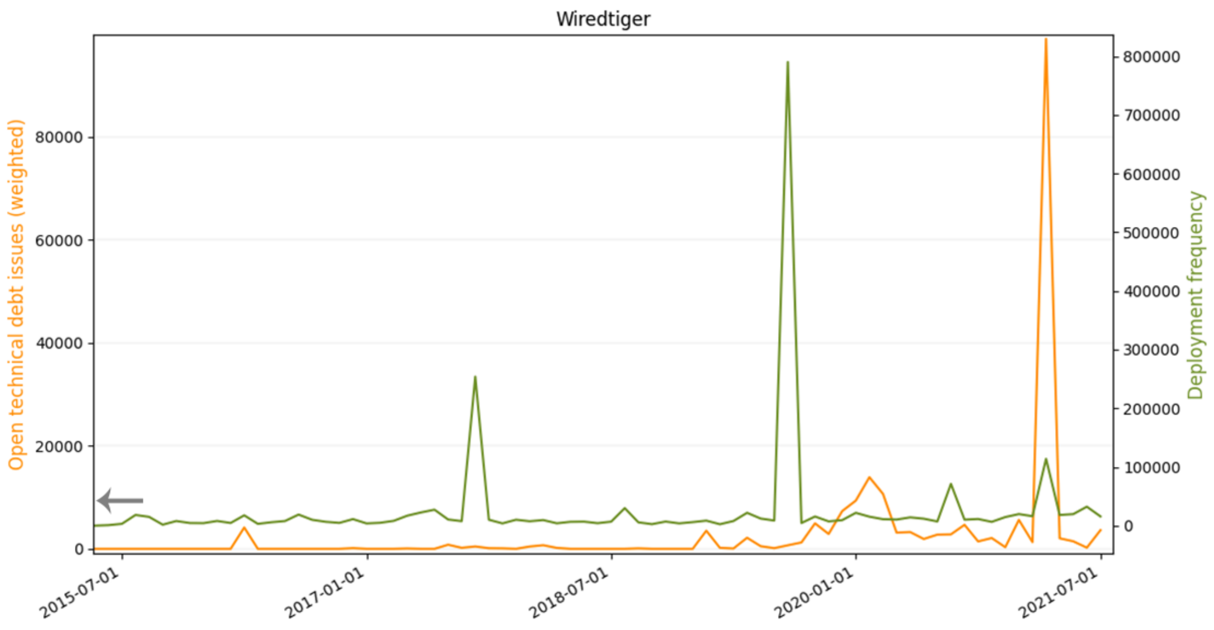


Figure 4.9: Line chart of deployment frequency and technical debt on a monthly basis for Wiredtiger. N.B., there are two different y-axes.

4.3.5 Lead time for changes line charts

Line charts for the lead time for changes for each of the OSS projects can be seen in this section. The charts feature lead time for changes compared to the technical debt over time on a monthly basis. Likewise to the deployment frequency, the monthly distribution can be seen from the dates on the x-axis that is formatted with a date order of YY/MM/DD.

Furthermore, just as it was with the deployment frequency, the y-axes for the lead time for changes and TD can be different. The y-axis for the opened TD issues can be seen on the left y-axis, and the lead time for changes (lead time per change) on the right. Chart seen in Fig. 4.10 is for Beam, Fig. 4.11 is for Flink, Fig. 4.12 is for Sakai, Fig. 4.13 is for Wildfly and the chart in Fig. 4.14 for WiredTiger.

The opened TD issues weighted represents the amount of TD that has been opened per month, weighted by their size. The lead for changes or lead time per change is the total amount of lead time for changes for a whole month. I.e., the sum of all lead time per change for a specific month.

As it was with the line charts for the deployment frequency, some of the earliest activity in certain OSS projects is excluded. The reason for excluding this from the charts is also that it excludes minimal activity from the projects, which makes it easier to represent the charts. A little arrow in the chart will indicate that the project had earlier activity.

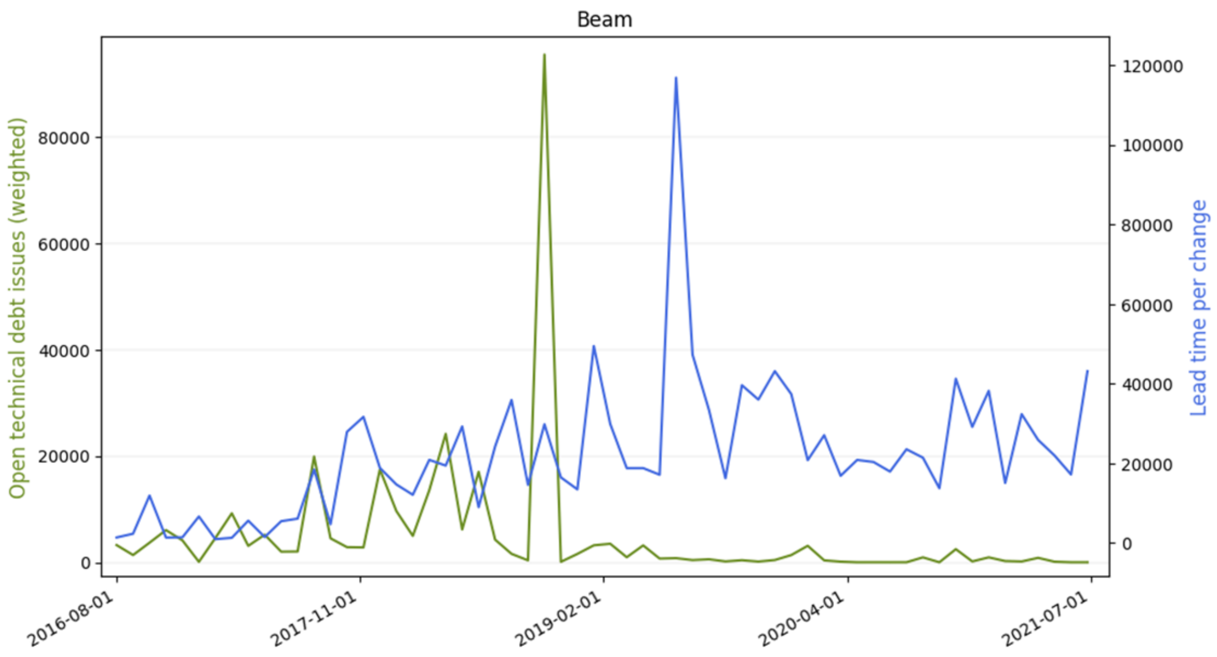


Figure 4.10: Line chart of lead time for changes and technical debt on a monthly basis for Beam. N.B., there are two different y-axes.

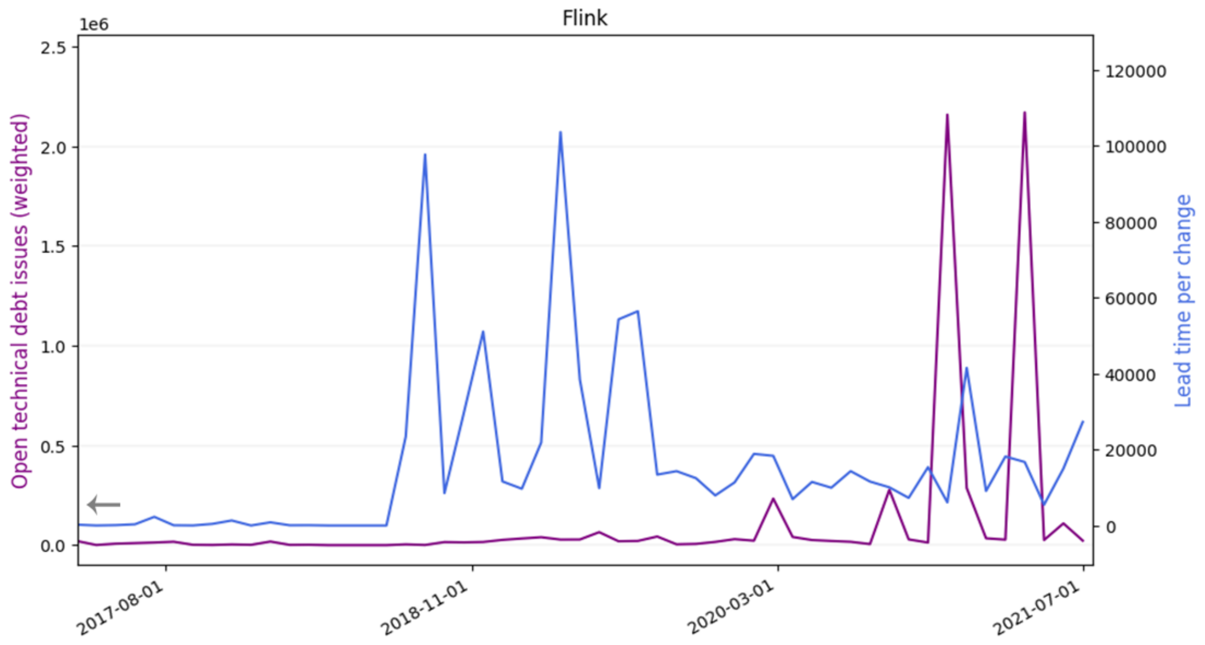


Figure 4.11: Line chart of lead time for changes and technical debt on a monthly basis for Flink. N.B., there are two different y-axes.

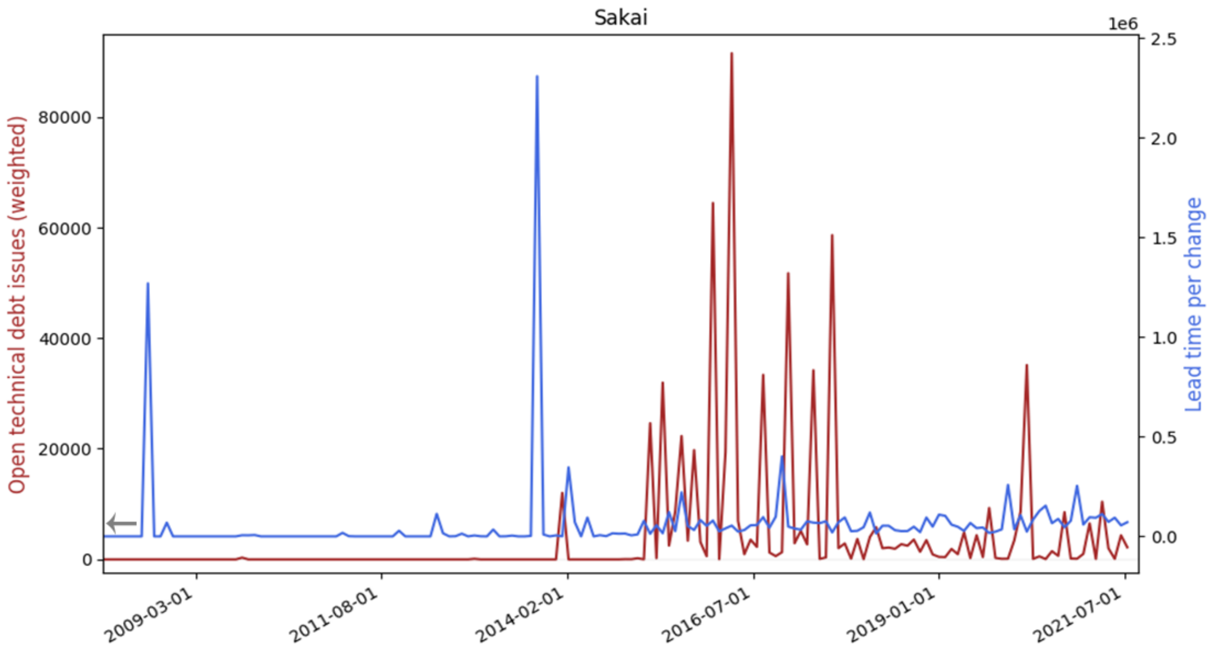


Figure 4.12: Line chart of lead time for changes and technical debt on a monthly basis for Sakai. N.B., there are two different y-axes.

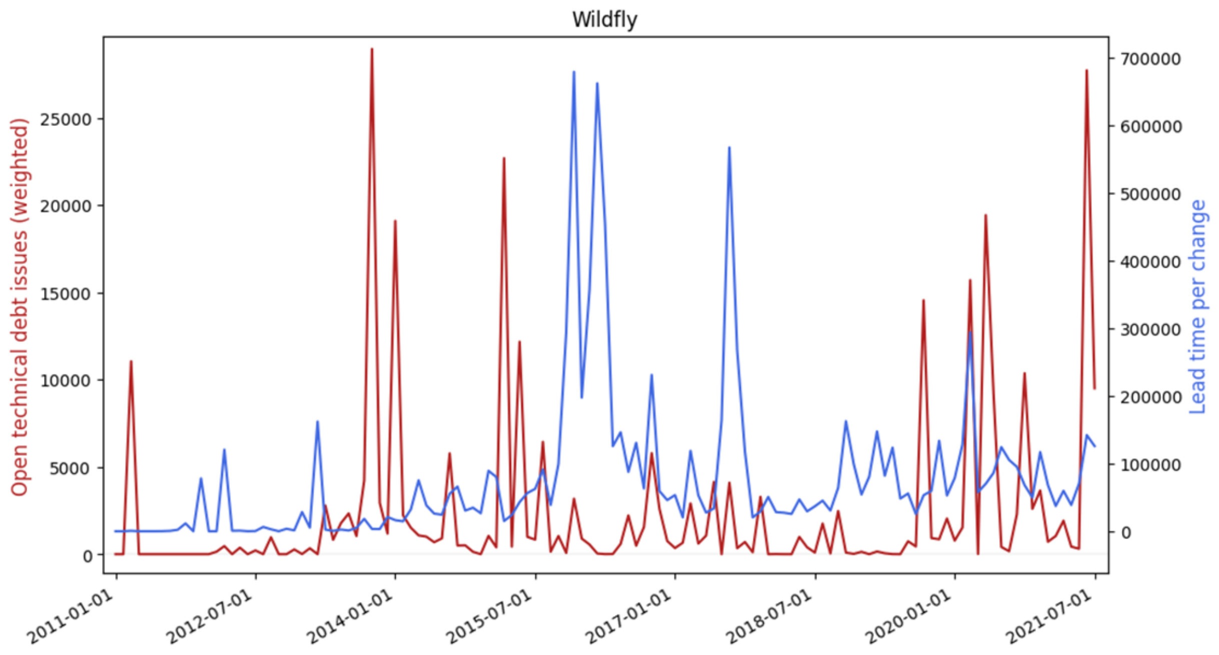


Figure 4.13: Line chart of lead time for changes and technical debt on a monthly basis for Wildfly. N.B., there are two different y-axes.

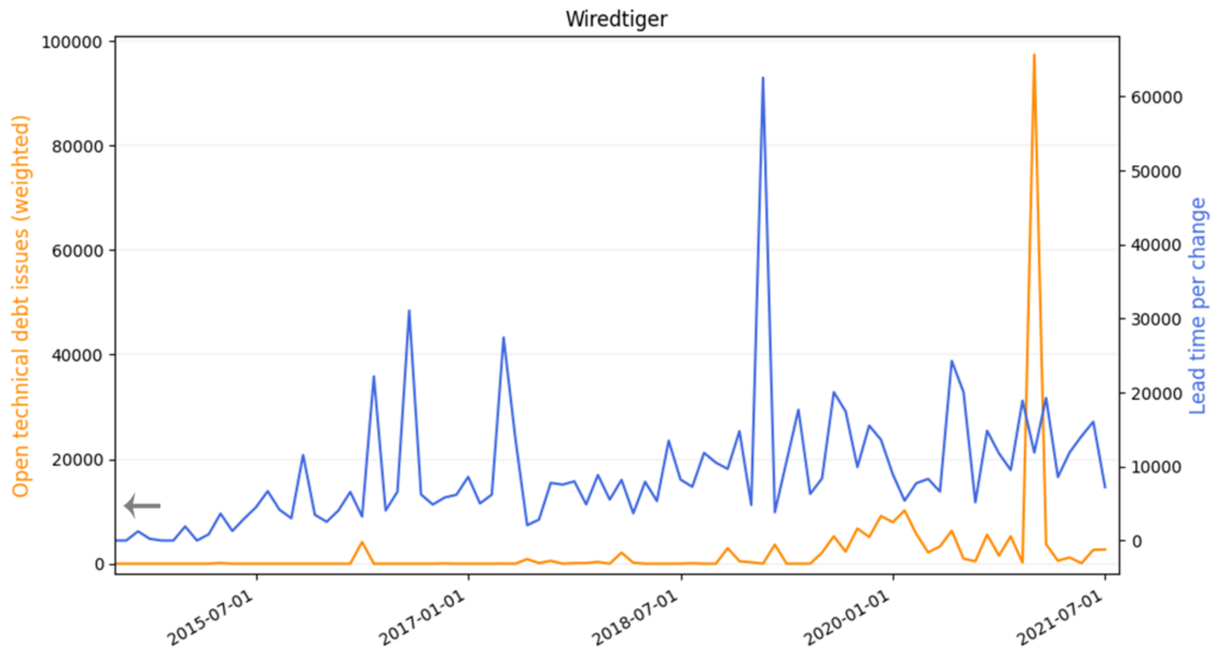


Figure 4.14: Line chart of lead time for changes and technical debt on a monthly basis for Wiredtiger. N.B., there are two different y-axes.

4.3.6 Deployment frequency scatter plots

The scatter plots for the comparison between deployment frequency and open TD issues are presented in this section. Just like it was with the line charts, the comparison is based on a monthly distribution. In order to make the data more presentable, the log transformation of both TD and deployment frequency has been taken. Distributing the data using a log transform will represent the data as a more normally distributed form. Correlations from Spearman's ρ and Kendall's τ are invariant for transformations that are monotone and are thus not affected.

In addition to the scatter plot, fitted lines has been applied to the plots to represent the relationship between the variables. The fitted lines, which can be seen in the plots as an order of either 1, 2 or 3, includes lines for linear regression, an quadratic function and cubic function of the data. Scatter plot for Beam can be seen in Fig. 4.15, Flink in 4.16, Sakai in 4.17, Wildfly in 4.18 and Wiredtiger in 4.19.

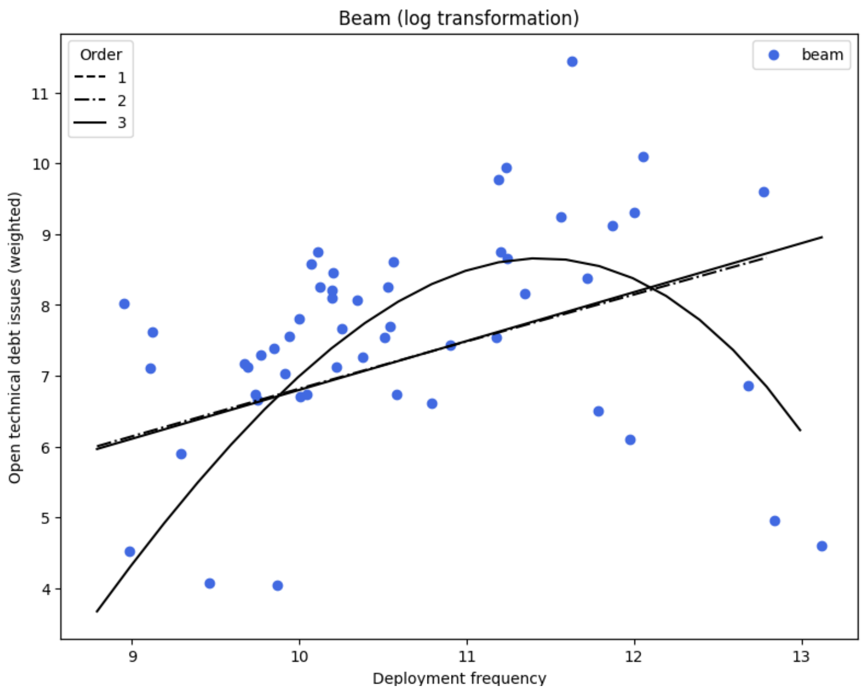


Figure 4.15: Scatter plot of deployment frequency and technical debt on a monthly basis for Beam.

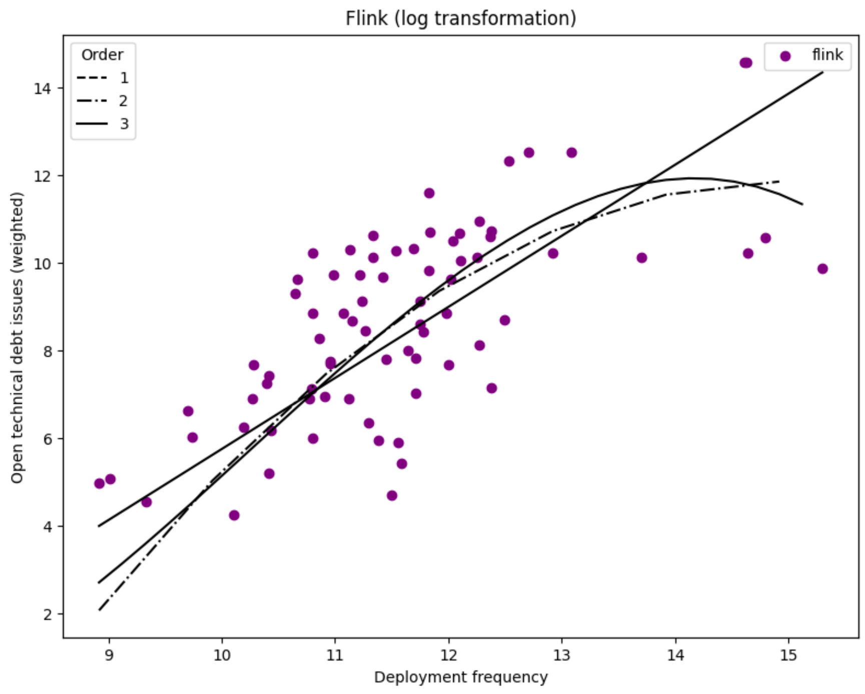


Figure 4.16: Scatter plot of deployment frequency and technical debt on a monthly basis for Flink.

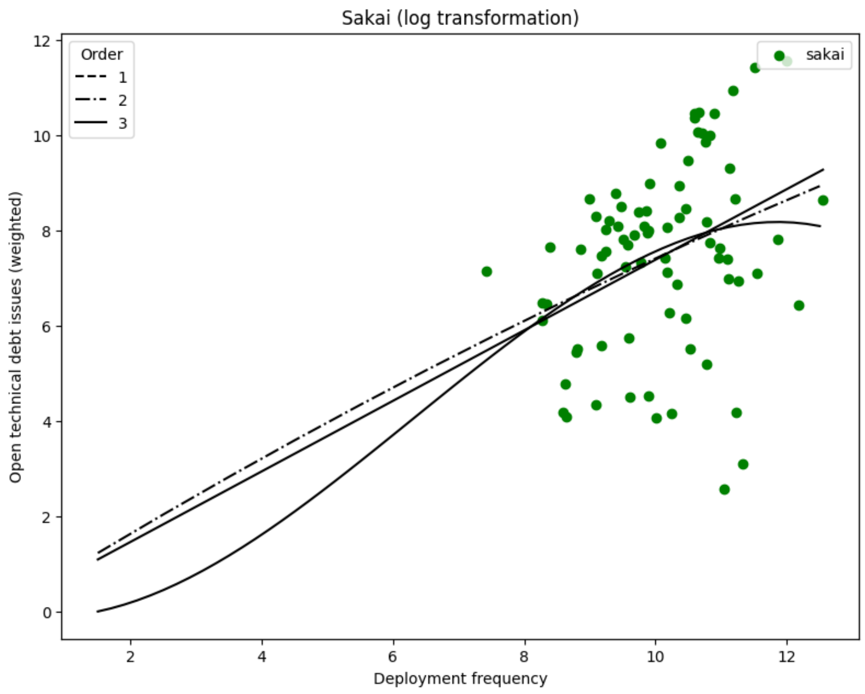


Figure 4.17: Scatter plot of deployment frequency and technical debt on a monthly basis for Sakai.

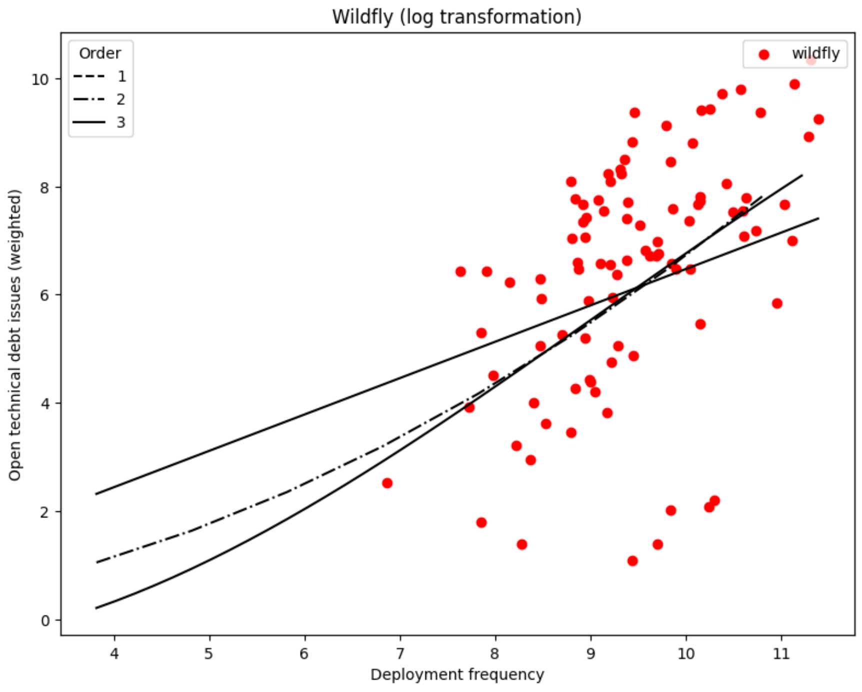


Figure 4.18: Scatter plot of deployment frequency and technical debt on a monthly basis for Wildfly.

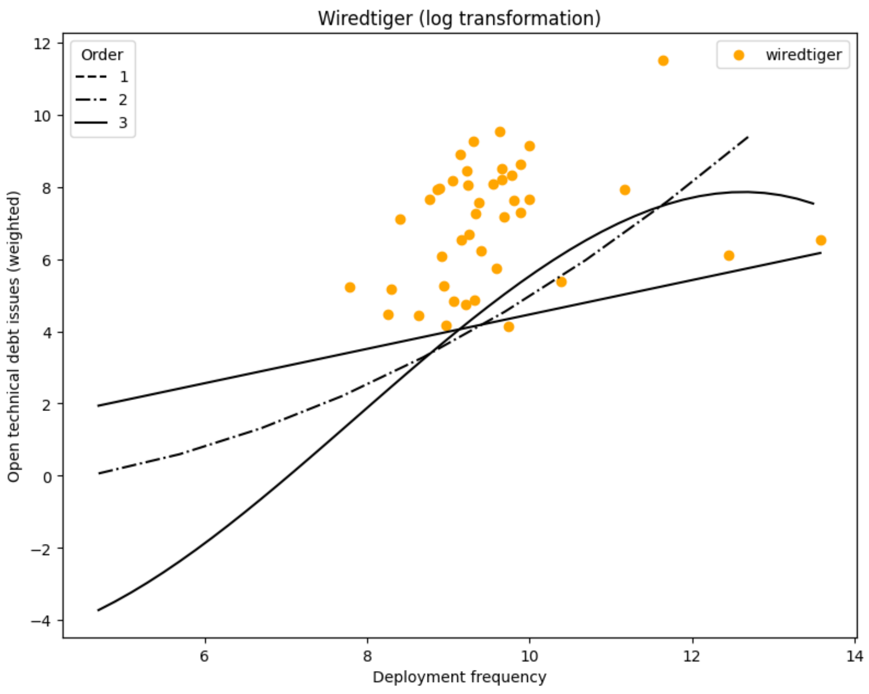


Figure 4.19: Scatter plot of deployment frequency and technical debt on a monthly basis for WiredTiger.

4.3.7 Lead time for changes scatter plots

This section presents the scatter plots for lead time for changes, also seen as lead time per change. Likewise to the plots for deployment frequency, the lead times for changes and open TD issues weighted are distributed on a monthly basis. The log transform of the variables has been calculated as a way to transform the data to a more normally distributed form.

As is the case for the previous scatter plots, fitted lines has been applied to the scatter plots for lead time for changes. The fitted lines can be seen in an order of either 1, 2 or 3. The scatter plot for Beam can be seen in Fig. 4.20, Flink in 4.21, Sakai in 4.22, Wildfly in 4.23 and WiredTiger in 4.24.

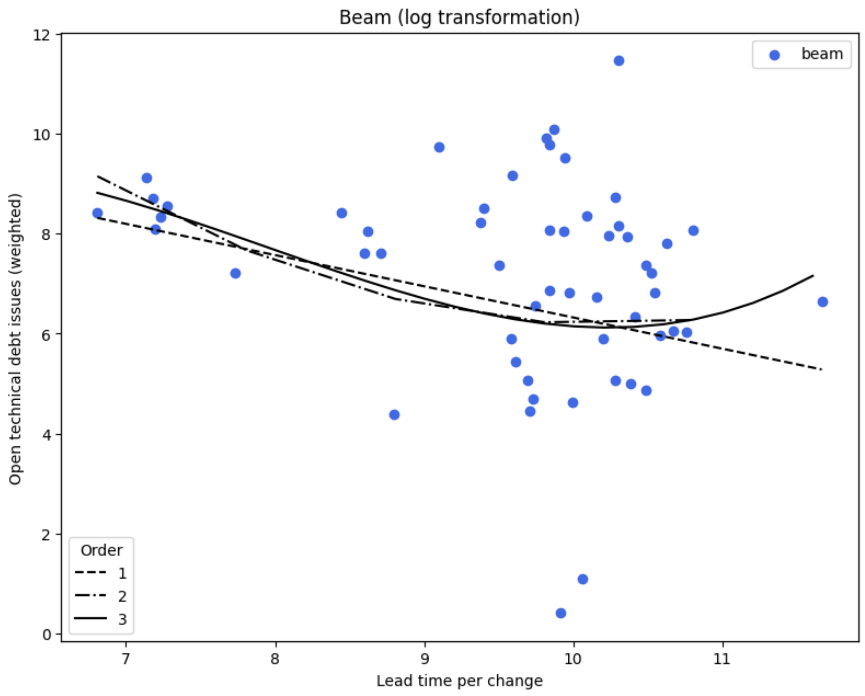


Figure 4.20: Scatter plot of lead time for changes and technical debt on a monthly basis for Beam.

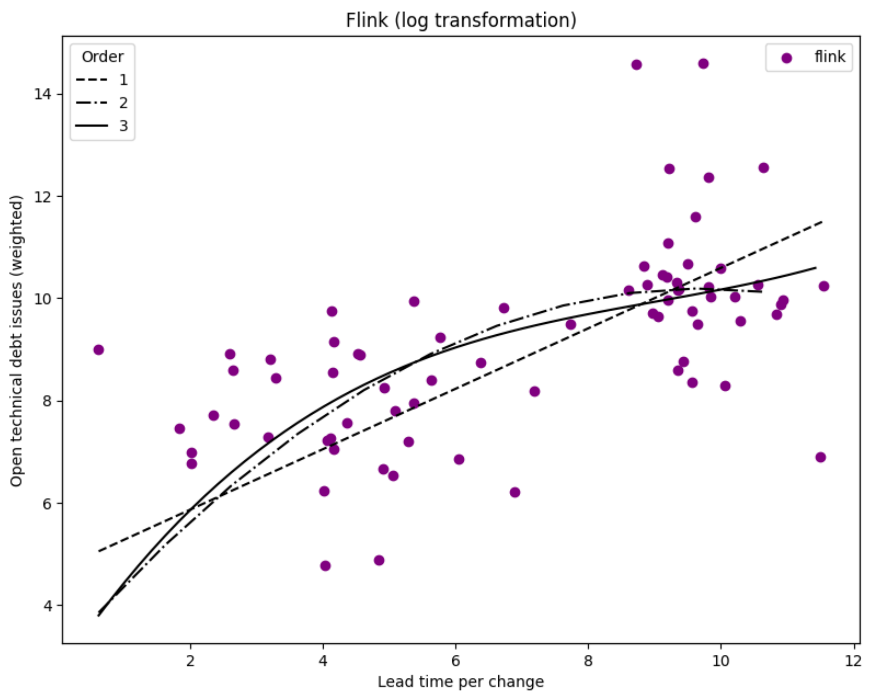


Figure 4.21: Scatter plot of lead time for changes and technical debt on a monthly basis for Flink.

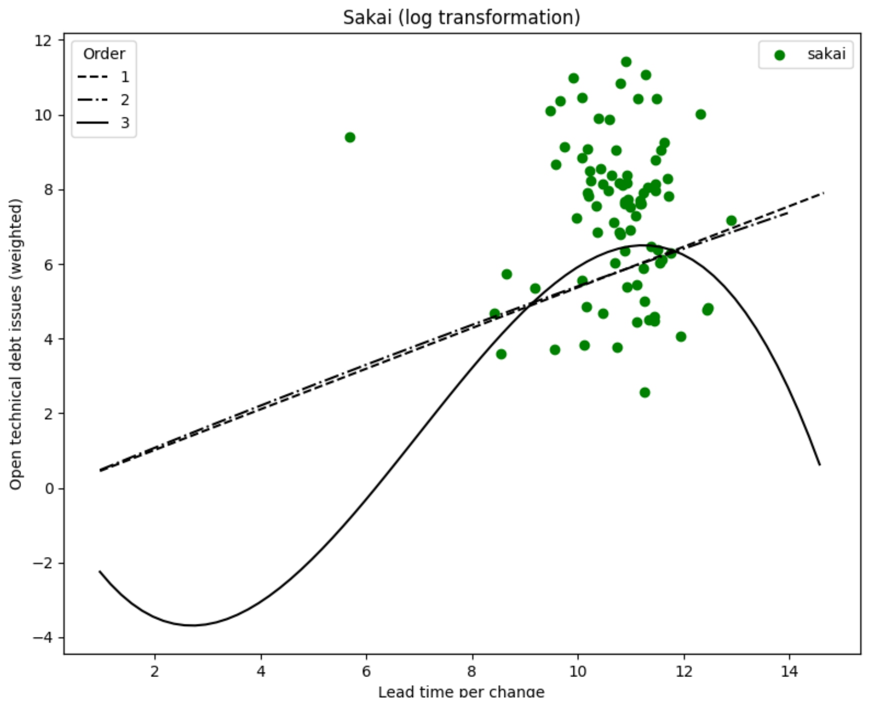


Figure 4.22: Scatter plot of lead time for changes and technical debt on a monthly basis for Sakai.

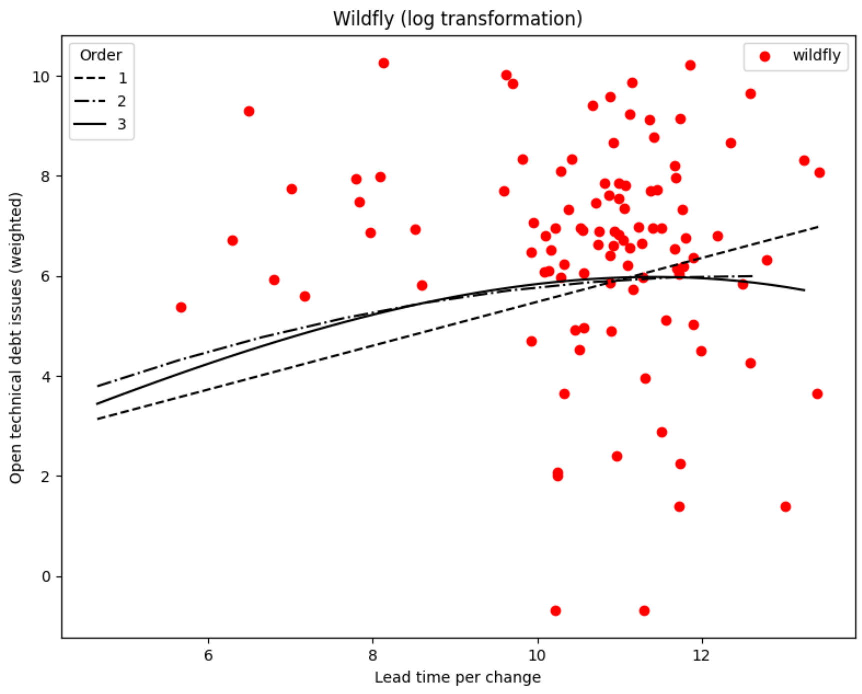


Figure 4.23: Scatter plot of lead time for changes and technical debt on a monthly basis for Wildfly.

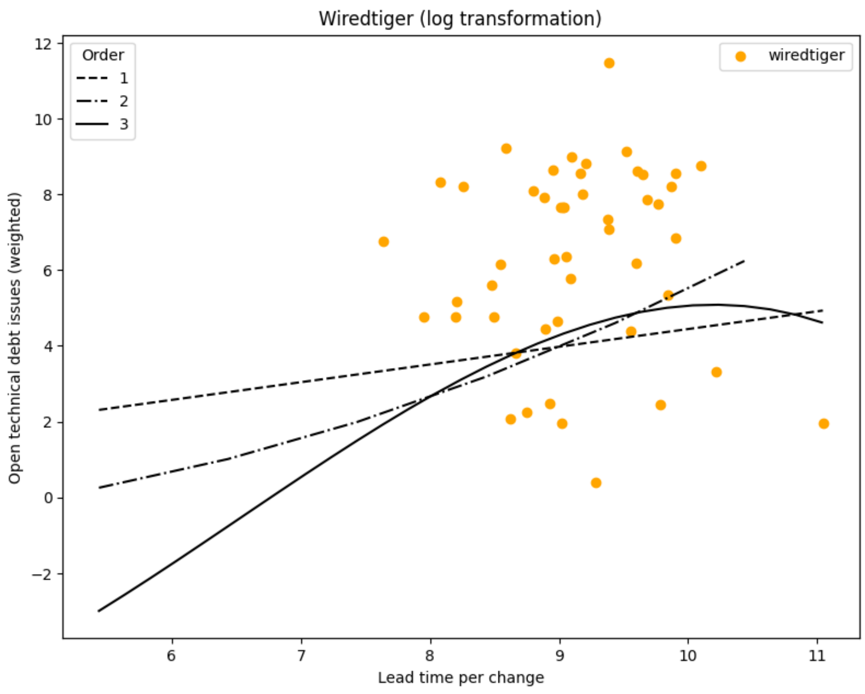


Figure 4.24: Scatter plot of lead time for changes and technical debt on a monthly basis for WiredTiger.

Chapter 5

Discussion

This chapter summarizes the key findings and interprets the results from the research questions. Both research questions are divided into separate sections, which follow the same structure as in the results chapter. First, for every section, the research questions will be restarted. Then, the key findings and the subsequent interpretation of the results are given. Lastly, the limitations of the study, threats to its validity, implications, and recommendations for future work will be presented.

5.1 RQ1 quantifying how TD evolves from developer discussions

The first research question, namely RQ1, asks about how NLP could be used to quantify TD issues from developer discussions, which included identifying how it evolves out of them. This is something that would further refine work carried out by a study from Ozkaya et. al [37]. In their study, they used NLP and ML to identify TD issues from discussions in the Chromium project. However, as they stated in their study, their work fell short of creating an oracle for this type of approach, where they concluded that there was a greater need for looking into improvements, such as increasing the accuracy of the classifications and bettering the feature engineering.

Asking this question thus meant that the limitations of this study had to be addressed. When doing so, the research question also tried to see if this approach could be extended to a larger scale. This would then mean that the classification method had to be generalized so that it could be applied to projects with differences in both technical and organizational varieties. In this case, five different OSS projects featuring a diverse variety of characteristics were selected and used. These can be seen in Table 3.1.

Data from these projects were then collected from Jira and Github. Before being prepared for data pre-processing, it would be possible to use NLP with ML models to classify the developer discussions from the data. This would then make it so that discussions would be classified as either TD- or non-TD issues, and subsequently, be linked to their respective discussions in Jira and code in Github. The ML models that were tested for classifying the data, had to be trained using supervised learning with a dataset that was separately built up. Relying completely on a dataset based on labels from Ozkaya et. al [37], failed to generalize for the technical and organizational varieties in the OSS projects.

Choosing to combine NLP with ML for detecting TD, has been proven to give good results from other studies. This was explained in the background of this thesis, where it was mentioned that studies such as Maldonado et al. [48] and Zhongxin et al. [32], have previously looked into how SATD could be detected from source code comments. Although this is different from what this thesis tries to answer. The findings from studies such as these, as well as Ozkaya et. al [37], seem to point toward NLP and ML is a good option for answering a question like RQ1.

Further, as the results chapter has outlined, the classification model that was ultimately chosen was the logistic regression classifier. The classification results from this can be seen in Table 4.6. Not only did this classifier achieve the best f1-score in comparison to the other models that were tested, but when analyzing the words with the top TF-IDF score, seen in Table 4.9, the words from this model seemed to make more sense than the other type of model that was tested. With the latter being the RNN model, which had been trained using deep learning, and word embeddings, as well as the dataset that had been separately built up.

In total, there have been three different ML models tested for RQ1. These include an RNN-, multinomial Naïve Bayes- and logistic regression classifier. The Naïve Bayes got a overall f1-score of 0.65, seen in Table 4.7. The RNN a 0.72, seen in 4.10, and the logistic regression classifier a score of 0.74. Both the Naïve Bayes- and logistic regression classifier have been constructed using TF-IDF, and thus shared the same scores for the top words. The RNN on the other hand used deep learning and word embeddings, and would therefore give different weighings to the words in the data. This resulted in top words that were significantly different than the other models, as can be seen in Table 4.12.

When inspecting the top words from the RNN, it happens to highlight some unexpected findings and interesting results. For example, among the top words, there were more diffuse words such as “rm”, “portalhanderexception”, “fa”, and “addsqlfunctionmethod”, as well as numbers including “459”, “968” and “430886993”. This is compared to the TF-IDF top words, such as “job”, “page”, “issue”, “memory”, “fix” and “beam”, seemed to be much more oriented towards code. Thus suggesting that the RNN may have put more emphasis on code snippets inside the developer discussions, while the TF-IDF may have instead been more inclined to make predictions based on the actual conversations between the developers.

After using the logistic regression classifier on unseen data with a threshold of P > 0.75, it would make TD predictions that ranged from 10.3-17.06%. As can be seen in Table 4.8. These numbers are within what other studies suggest for SATD, such as Potdar et al. [42] that estimated between 2.4% - 31%. And Ozkaya et. al [37], that estimated roughly 16.1%. In addition to this, as a measure to further verify the predictions from the classifier. A sample of one hundred issues of the predictions from the model would be manually inspected and classified by another informatics master's student from the University of Oslo. Using the rubric from Bellomo et al. [8], the student matched 72 of the 100 predictions by performing a manual classification. This seems to correspond well with the scores one would estimate could come from the logistic regression model.

Furthermore, when it comes to answering RQ1 based on the results from the classifications and predictions, where the latter can be seen in Table 4.8. Then, the data from this suggests that identifying and quantifying TD from developer discussions can be completed using a combination of NLP and ML. This thesis achieved this by training the logistic regression classifier using an extensive dataset, which seemed to benefit from featuring a wide variety of organizational and technical differences.

Although the overall f1-score of this model was only 0.74, leaving room for more improvements. It was still able to not only obtain better performance scores compared to the one from Ozkaya et. al [37], but also generalize the classifications across five different OSS projects. As seen in Table 4.4, their model had a precision of 0.40 and a recall of 0.62 while the logistic regression classifier used for this thesis managed to achieve a precision of 0.76 and a recall of 0.73. This can be seen in Table 4.6. Thus, the logistic regression model is significantly better at identifying and subsequently quantifying TD issues from developer discussions.

In addition to this, when it comes to the method and processes used for collecting and preparing data. Then this made it so that both Jira and Github data would be linked together, where their relationship would be based on issue keys from Jira. Making it so that it would be possible to see how TD issues evolve. As both the methodology chapter and results section for the collected data explains, the TD issues will be associated with timestamps and other data. Thus, both the former and latter could be used for other purposes, such as looking at how TD evolves from these discussions and links to other data, as well as how the TD- and non-TD issues compare with the projects as a whole.

However, as was just pointed out by the results from the logistic regression model. There is room for improvement. Including further increasing the accuracy of the classifications, looking even more into feature engineering, and possibly also increasing the size of the dataset used for supervised learning. Considering that the dataset was relatively small, containing only 1,501 issues. Increasing the dataset could then potentially have a significant impact on training a classification model. Furthermore, when it comes to featuring engineering and increasing the accuracy of the classifications. It may also be interesting to further address what the RNN model has called attention to, which is to look more into how code snippets within the developer discussions can affect the classification.

Having this systematic method for quantifying TD, which includes being able to see how they evolve from developer discussions, could potentially be an invaluable measure for decision-making related to TD prioritization. When looking at TD this way, compared to more frequently used methods like code inspection with tools such as SonarQube. As it might be used as a new way for both monitoring and dealing with TD issues in large projects, as well as identify TD issues that would otherwise be hard to uncover with code inspection. Including architectural design concerns, suboptimal development choices, and awareness of up-front solutions that are below par, where it would also be possible to track individual issues and follow how they evolve, both in relation to the discussion and the associated code.

5.2 RQ2 using quantified TD with DevOps metrics to obtain insight

The second research question asks about how the evolution of the quantified TD, which is a result of RQ1, can be used together with the DevOps metrics to give insight into projects. That is, RQ2 tries to see how these two compare through correlation analysis before the eventual results from this can be used as insight. These results have also then been subsequently visualized. Further, since this thesis has quantified TD from five different OSS projects, seen in Table 3.1, where each features its own technical and organizational varieties. The insight that comes from correlating the TD with the metrics, will be related to each project respectively.

The specific DevOps metrics that have been measured are the ones for velocity. To be more precise, the lead time for changes and deployment frequency. These metrics stand in contrast to the stability metrics, namely the change failure rate and time to restore service. Further, to justify any confounding factors that size may have on the projects, both the metrics and quantified TD have also been correlated with the size of each project. This was selected as the OSS project's Github repository size. I.e., the size measurement is the total number of lines for the repository, including the line additions, deletions, and modifications.

The basis of RQ2 is to see how TD correlates with the velocity of software development, as well as further add to work carried out by Lenarduzzi et al. [28]. They proposed a data-driven approach for TD prioritization, where they suggested that this may relate TD to its interest. In their study, they compared TD from numerous projects with their lead time. The TD from these projects came from the Technical Debt Dataset [25]. As mentioned in the background of this thesis, this dataset has used SonarQube to analyze projects as a way to quantify TD. I.e., code inspection has determined the amount of TD in the projects. The lead time, on the other hand, was measured as the time it took to resolve Jira issues.

This is different from what this thesis tries to answer. Although RQ2 explores a data-driven approach for analyzing TD with the performance of software development, in this case, the velocity. Both the quantified TD and metrics are different, as described in Eq 3.14 the quantified TD is a product of both the number and size of the TD issues. More specifically for the correlation analysis, the quantified TD was selected as the number of open TD issues that have been weighted on a monthly basis. This relationship is described in Eq 3.14. Thus, the quantified TD is different from how TD has traditionally been quantified with code inspection [6, 26]. Choosing specifically to correlate with the number of open TD issues, has been done to see how open TD issues can affect the project in terms of the metrics.

Further, the metric here for lead time for changes is also different. As compared to their study, this thesis has used mapped PRs to calculate the lead time for changes. This was also then further specified by converting it into lead time per change, as this would additionally describe the metric. Measuring this lead time for changes has thus been achieved by calculating all the lead times per change for each PR, which would then be associated with their respective Jira issues. This relationship is summarized with Eq. 3.13. Moreover, when it comes to the measurement for the second DevOps metric, namely the deployment frequency, then this was measured as the deployed value over a specific period. This results in a frequency of deployed value, where the deployed value would be equal to the size of the successful deployments as described in Section 3.4.3.

The specific period that was selected was on a monthly basis. This period was also used for all of the analysis for RQ2, as well as its correlations. Including the deployment frequency, lead time for changes, quantified TD, and size correlations. The same applies to the visualization, where the line charts and scatter plots have also been distributed on a monthly basis. However, there is one exception to this which is that the first five and last five months of each project have been excluded from both the correlation analysis and visualization. As explained in the methodology chapter, excluding the first five will try to ensure that the projects are mature. Hence, may have had the chance to accumulate TD [42]. The last five are removed because it is likely that open TD issues might be still open. Hence, keeping them will not give a good estimation.

From the correlation analysis results, it can be seen that there are some mixed results, some unexpected findings, and interesting insights. This, along with how they may potentially bring insight into how TD is prioritized and fixed, will be further discussed in this section of the thesis. However, since the analysis is rather extensive, each part of the discussion for the correlation analysis has been split up into its respective sections.

Having a method such as this for a data-driven approach when dealing with TD, may have the potential to lead to significant changes in how TD is ultimately prioritized. As Lenarduzzi et al. [28] points out, such an approach may be used for estimating the interest of the TD, by establishing an association with the TD and how it influences performance.

5.2.1 Lead time for changes correlated with TD

The lead time for changes correlated with open TD issues can be seen in Table 4.13. While the Pearson's correlation coefficient r seems to not indicate any meaningful correlation between these two. Both the correlations for Spearman's rank correlation ρ and Kendall's rank correlation coefficient τ seem to give meaningful correlations. As seen from ρ, there seems to be a monotonic relationship being present. The projects Flink, Sakai, and WiredTiger had a strong positive relationship with a correlation degree of 0.685, 0.690, and 0.602 respectively. Wildfly got a weak positive relationship with 0.245. However, Beam seems to result in almost the exact opposite of the latter, as the correlation is a weak negative relationship of -0.259, thus being a deviation from the rest of the projects.

Section 4.3.5 visualizes this relationship in the form of line charts, where the variables can be seen distributed over the individual months of every project. From the charts, it is also possible to see the monotonic relationship for each of the projects. With Flink, Sakai, and WiredTiger it is especially noticeable that periods with a high amount of lead time per change (lead time for changes), seem to result in more open TD issues. This data may suggest that a period where it takes a long time to deliver code, will result in more open TD issues later on. It could also mean that when something doesn't work, developers start discussing TD issues and then fix them.

For instance, with Flink seen in Fig. 4.11, the period that spans from the middle of the year 2018 and early 2020, seems to indicate that there is a high lead time per change for this duration. I.e., it takes a long time for code to be pushed into production. Furthermore, as the monotonic relationship suggests, this is shortly followed by a period with an increased amount of open TD issues. As seen from Fig. 4.11, there seem to be a lot of new TD issues being opened in the timespan of late 2020 to late 2021.

The same applies to the projects Sakai, WiredTiger, and partly also Wildfly. As these projects also result in a positive monotonic correlation degree. However, as mentioned, Beam was a deviation from the rest of the projects, as its correlation happen to result in a negative relationship. From its line chart, seen in Fig. 4.10, it can be seen that it has the opposite happening compared to the other projects. This may almost suggest that a period with a great number of open TD issues, which would later be resolved, will result in a subsequent period with a high lead time for changes.

However, interpreting the correlation degrees purely on its own may not give the full picture. As it is important to note that the quantified TD is open TD issues that are based on developer discussions. The TD will therefore represent developers who have expressed TD based on the sentiment of the discussions on the issues. This can, as explained in RQ1, be either directly or indirectly expressed. Further, the charts also interestingly show the periods with high amounts of resolved TD issues are not the same periods where there is a high lead time for changes. As seen from Section 4.3.5, the latter is the case for all five of the different projects.

This unexpected finding may suggest those periods where developers experience it as slow to push to production. In other words, when there is a high lead time for changes, may result in developers shifting their focus from pushing code, perhaps because it takes longer, and then over to start discussing TD issues more frequently. After the developers have either directly or indirectly become aware of the accumulated TD and subsequently resolved the issues, then the lead time for changes seems to decrease. This suggests that the result of this will give a better production capacity, where the developers are faster at delivering non-TD (e.g., updates, features, and so forth) issues to the projects.

The same reasoning may potentially also be applied to Beam, seen in Fig. 4.10. Although the chart for this project seems to indicate the opposite of this, where resolving TD issues results in a high lead time for changes. The same period with high lead time seems to somewhat decline after some time has passed from when the TD issues have been resolved. This could suggest that fixing the TD issues has caused a dispersion in the project, where there would be a lingering effect on the lead time for changes. The same phenomena can also be observed in Wildfly, seen in Fig. 4.13. In that TD issues are frequently discussed before a shorter period of high lead time for changes is followed, and then subsequently followed with what seems to be a decline in lead time for changes.

Further, for both Beam and Wildfly, it can be pointed out that it seems to be more uncertainty around what could have prompted developers to suddenly discuss TD issues in such high numbers. Compared to the other projects, namely Flink, Sakai, and WiredTiger, where the periods with a high lead time for changes can be interpreted as the reason why TD issues are suddenly being discussed in large numbers.

All in all, this may suggest that the projects have other things in common than what the correlation degrees represent. The same also goes for their scatter plots seen in Section 4.3.7, where the relationship between the lead time for changes and open TD issues doesn't seem to be so clear. In that case, this will be further supported by what Lenarduzzi et al. [28] discovered in their study, which was that they couldn't find any meaningful correlations between their way of quantifying TD and lead time. Although the ρ correlations initially gave significant results for most of the projects. All the trends from the projects, when they were spread across a timeline with the line charts, seem to indicate that there could be a more complicated relationship between the variables.

This correlation analysis may therefore be limiting, in that there doesn't seem to be any conclusive answer about the findings for all five OSS projects. However, the correlations and the analysis of the data seem to reveal some significant results and interesting findings. Any future work could therefore be to look into what this thesis has highlighted. Including, for example, a further assessment of why certain projects resulted in a high positive ρ correlation, while a project like Beam did not.

5.2.2 Deployment frequency correlated with TD

Contrary to lead time for changes, the deployment frequency correlated with open TD issues seems to result in linear correlations. As seen in Table 4.14, the Pearson's correlation coefficient r ranges from a positive very weak correlation to a strong one for the different OSS projects. Where Beam, Flink, Sakai, Wildfly, and WiredTiger got 0.154, 0.467, 0.450, 0.610, and 0.119 respectively. The p-values for Flink, Sakai, and Wildfly are also significant. This may then initially suggest that there does exist a linear correlation between the deployed frequency and open TD issues.

Further, as seen with both the correlations for Spearman's rank correlation ρ and Kendall's rank correlation coefficient τ, these two seem to also give strong correlations. For ρ specifically, the degrees range from a strong to a very strong correlation where Beam, Flink, Sakai, Wildfly, and WiredTiger resulted in 0.543, 0.729, 0.917, 0.716, and 0.661 respectively. Based on p-values for all the projects, the monotonic relationship in the projects also seems to be very significant.

The monotonic relationship for the OSS projects can be seen in the scatter plots in Section 4.3.6. However, this very same relationship seems to be more unclear when looking at the correlations from a timeline perspective, where the line charts seen in Section 4.3.4 represents this. Although, from these line charts, the relationship between the deployment frequency and TD issues seems to point to some interesting insights.

For example, periods with a high amount of deployment frequency. I.e., periods where there is plenty of value delivered to the projects, seem to overlap with periods that have a lot of opened TD issues. This can for instance be seen in the line chart for Wildfly with Fig. 4.8, where it is visible that periods with a high deployment frequency, such as late 2019 to the middle of 2021, also have a lot of opened TD issues in the same duration. The same can be said for Flink, seen in Fig. 4.6, where the period of late 2020 to late 2021 seems to have a high amount of deployment frequency, which is overlapping with a great amount of opened TD issues.

One could then, if the correlation is taken literally, assume that it would mean that the more TD there is in a project, the more value is being delivered. On the other hand, it could also mean that the faster developers deliver value, the more TD is accumulated. However, as pointed out for the correlations between lead time for changes and open TD, this relationship shouldn't necessarily be interpreted literally. The quantified TD or in other words, the open TD issues, will in this case be based on the sentiment from developer discussions. It is therefore the developers themselves that have expressed some form of concern for TD, either directly or indirectly.

This can then ultimately suggest that when the developers become aware of their accumulated TD, hence the high amount of TD issues being opened, that this would also seem to be the time when developers happen to deliver a lot of deployed value. Thus, the result of this is that there is an overlap between the open TD issues and the deployment frequency. Further, if that happens to be true, then it may be fitting to assume that making developers aware of any TD that have incurred in the project, could potentially also lead to more value being deployed to the project.

Nonetheless, there will still be some uncertainty surrounding a possible conclusion here. As this thesis looks at it from the perspective of a multiple-case study, rather than going into more detail about what might be happening here. Further works may then be to look more closely at what is happening here, possibly by going more in-depth.

5.2.3 Project size

This part of the discussion covers how the project size may have affected the various variables. To systematically summarize the key findings and the subsequent interpretation, the variables have been divided into separate sections.

Lead time for changes correlated with size

The results from correlating project size with a lead time for changes can be seen in Table 4.15. From the table, it is visible that there is a negative very weak linear relationship between the two variables. This can be seen with the Pearson's correlation coefficient r, which measures linear correlations that seems to indicate very weak correlations for most of the OSS projects.

Further, both Spearman's rank correlation ρ and Kendall's rank correlation coefficient τ seem to result in negative weak to negative moderate correlations. Only the Flink project is an exception to this. For ρ, then p-values seem to be significant. Intriguingly, this may then suggest that as the project gets bigger, the smaller the lead time for changes becomes. Similarly, the smaller a project becomes, the higher the lead time for changes becomes. This can then initially be interpreted as that it is easier to make changes to a smaller project. I.e., the bigger a project gets, the harder there is to make changes to it, which then results in less lead time for changes. However, how this would correspond with the results from lead time for changes being correlated with TD is uncertain.

Deployment frequency correlated with size

From Table 4.16 the results from correlating deployment frequency with size can be seen. As seen from the table, there are mixed results from all correlation tests. For example, Flink got a ρ strong positive correlation of 0.691, but WiredTiger resulted in a negative moderate correlation of -0.292. Further, Beam got a ρ of 0.176 whilst Wildfly resulted in 0.089. Thus interpreting the results and relating them to the deployment frequency is difficult. As the results seem to indicate that the OSS projects are very different from each other.

Technical debt correlated with size

Likewise to the deployment frequency correlated with size, the technical debt correlated with size also gave mixed results. Table 4.17 summarizes this. In the table, it can be seen that there seems to be some linear correlation r between the OSS projects, except WiredTiger which got a very low correlation degree. However, based on the p-values, only Flink seems to have significant results.

Further, as seen with Spearman's rank correlation ρ and Kendall's rank correlation coefficient τ, these seem to also give mixed results, where three projects, namely Sakai, Wildfly, and WiredTiger, seem to result in a negative correlation. Whilst the projects Beam and Flink seem to have a positive one. Interpreting these results may therefore be difficult from this data alone, as the projects don't seem to have many similarities.

Moreover, this may then suggest that project size is not a substantial confounding factor. In that, the variables when they were correlated with the size, did not seem to indicate any meaningful correlations.

5.3 Implications for practice

This section provides future recommendations for practitioners, which are based on the overall findings from the thesis. Some of the more specific details for each of the RQs can be found in their respective discussions.

First of all, it looks like NLP and ML can be used for classifying TD issues based on the sentiment from their associated discussions on a large scale. This approach also seems to some extent generalize to projects with technical and organizational varieties. Using the quantified TD obtained from this approach can then further be combined with other data-driven approaches for insight. Including studying the relationship between the quantified TD and DevOps metrics, as it was done in this thesis.

The findings from this approach for quantifying TD may therefore be used as an alternative and proactive measure for decision-making related to TD prioritization. In addition, the approach may also be used for keeping track of issues in large projects, where it would be possible to identify issues that could otherwise be hard to do with code inspection. This can then be such as architectural design concerns, awareness of up-front solutions that are below par, and suboptimal development choices. Additionally, the quantified TD may also be used together with other metrics, where the relationship between the TD and metrics can be further investigated.

Further, correlating and visualizing TD with DevOps metrics may also prove to be useful. As it can be used to see trends in the software project management related to TD prioritization. Relating performance metrics, such as the DevOps metrics, to TD issues may also specifically give insight into the relationship between TD and performance.

5.4 Implications for research

The final model that quantified TD with NLP and ML for this thesis was supported by work carried out by Ozkaya et. al [37]. Firstly, this thesis found that their approach could be used for detecting TD issues based on developer discussions at a large scale, just as they concluded. Secondly, as they expressed, the model was found to be useful for identifying features from discussions that seemed to be strongly associated with TD issues. Lastly, as they point out in their study, these features were found to be useful for further understanding how TD can be used and investigated.

However, relying completely on the sample data from their single-case study did not seem to generalize for the multiple-case study that was conducted for this thesis. To increase the classification accuracy of TD issues, this thesis had to further refine the sample dataset, the feature engineering, and the model. With this in mind, this thesis may therefore serve as a foundation for the modification of their theory and work.

Further, the background for investigating the relationship between TD and how it relates to performance measures has been partially supported by Lenarduzzi et al. [28]. Coupled with theories from other studies, which have shown that TD seems to have velocity implications for software in the long term, such as making it harder to add new features [4, 34]. In this thesis, it was specifically looked into how TD correlates with the velocity of software development performance, which would then be correlated with the velocity DevOps metrics defined by DORA [1].

Likewise to Lenarduzzi et al. [28], the findings from correlating open TD issues with a lead time for changes did not result in a definitive conclusion. However, interesting findings were made from analyzing the results of this thesis. Moreover, when it comes to correlating open TD issues with the deployment frequency, then this resulted in what other studies have

[1]https://cloud.google.com/devops/state-of-devops

concluded when it comes to TD having an impact on software performance. As the results from this indicate that the deployment frequency is affected positively by refactoring TD issues. In the case of this thesis, identifying open TD issues and then subsequently refactoring them, seemed to result in more deployed value being added to the projects.

5.5 Limitations

Manually labeling: Issues that have been manually labeled and selected for the dataset used in RQ1, have been based on a rubric from Bellomo et al. [8]. This exposes the manual labeling for a subjective judgment that can have significantly affected the outcome.

Construct validity: This thesis has for RQ1 relied on testing and evaluating the models with standard practices for ML, including k-fold cross-validation and train-test split. The classifications from the final model have also been further verified, by manually labeling and matching a random sample of one hundred issues. To avoid confirmation bias, another informatics master's student at the University of Oslo was tasked with this and used a rubric from Bellomo et al. [8].

The DevOps metrics measured for RQ2 relied on concepts from DORA [2] and were further defined as precise metrics that were proxies for the concepts. However, a threat to the construct validity is that the proxies might not fully represent the construct. For example, there is still uncertainty about the precision of measuring TD issues from developer discussions.

Internal validity: Issues manually labeled for the dataset used for training the models, have been made vulnerable to subjective judgment. Further, there have been made measures to mitigate threats of confounding factors in the correlation analysis. Both the quantified TD and DevOps metrics have been normalized based on issue size, as well as correlated with project size. However, a threat to the internal validity is that there could be other factors that have not been controlled for with these measures.

External validity: To increase the external validity of this thesis, a multiple-case study of five OSS projects was selected. Although the projects featured different technical and organizational varieties, it is still difficult to tell how the results may generalize to other contexts, such as projects that are not open source and that feature different technical varieties.

Reliability: To increase the reliability of this thesis, all the data for this thesis has been made available as supplemental materials. This may then be used so that the same results can be replicated. This includes raw data, the scripts, and the final data.

[2]https://cloud.google.com/devops/state-of-devops

Chapter 6

Conclusion

The first research question asked about how NLP can be applied to developer discussions to quantify the evolution of TD. This thesis found that NLP can be used with ML to classify TD issues based on the sentiment of developer discussions. Then, as the issues would be linked together based on relational data from Github and Jira, this thesis was able to relate the TD issues to timestamps and other data including code. This made it possible to quantify the evolution of TD issues, normalize the TD issues based on their size, and subsequently use it as a measure.

Such measures may be used to identify TD issues that could otherwise be difficult to detect with more common tools like code inspection. Also, the measure may be used for decision-making related to TD prioritization and other data-driven approaches. Additionally, it may also be used as a proactive measure to both track and monitor TD issues in large projects. However, one important drawback of this approach is that the TD issues are only detected when developers start discussing them. As opposed to static code analysis, where TD can be detected regardless of whether developers are aware of the TD or not.

The final model used for classifying issues further refined work carried out by Ozkaya et. al [37], which had previously used NLP and ML to detect TD from developer discussions in a single-case study. In this case, the final model for this thesis was able to both outperform theirs, as well as generalize for the five different OSS projects. As compared to their model precision of 0.40 and recall of 0.62, the final model for this thesis was able to achieve a precision of 0.76 and recall of 0.73.

The second research question aimed to correlate the quantified TD together with the DevOps metrics, so that this may be used to get insight into how TD correlates with velocity. This thesis did find some meaningful correlations between the TD and lead time for changes, but the results from this were uncertain. While most projects resulted in strong positive correlations, there was one project that diverged from this and got a weak negative correlation. The result of the correlation analysis may then suggest that there is a more complicated relationship between the two variables.

However, the findings from the correlation analysis did result in some interesting findings. Firstly, TD management seemed to occur as short-term bursts with high frequencies. Secondly, the TD management periods with the most amount of resolved TD issues did not seem to overlap with periods where there is a high lead time for changes. This may suggest that developers, when they experience it as slow to push to production, shift their focus to discussing, opening, and resolving TD issues. In that case, it will be consistent with how TD is perceived to have implications for software in the long term. As the observation from these periods may then ultimately suggest that if TD is not being actively managed, it will make it difficult to add new features and make it more costly to refactor.

Finally, correlating the TD with deployment frequency resulted in strong correlations for both linear and monotonic relationships. On one hand, the findings from analyzing the relationships seemed to suggest that developers benefit from acknowledging TD, as when developers started to discuss and open TD issues, the deployment frequency increased. On the other hand, it could also be that developers seem to prioritize TD issues after they have been identified and discussed, and as a result, TD issues are solved faster than other things such as adding features.

6.1 Future work

Several interesting findings have led to topics that can be further researched.

Firstly, the final model used to classify TD issues may serve as a reference point for future work. Despite being a significant improvement to previous work, there are still opportunities for further enhancements. This includes increasing the accuracy of correctly classifying the sentiment of developer discussions, so that it may provide an even more reliable way of quantifying the TD issues. Further, as observed from the RNN classification results, it may also be interesting to look into how code snippets within the developer discussions affect the classification. Additionally, it may be looked into how it is possible to differentiate between various types of TD.

Secondly, it might be interesting to see how the quantified TD can be used to further study the relationship between the DevOps velocity metrics. For instance, by going more in-depth with the observations that have been made in this thesis, such as what appears to be a more complicated relationship between the lead time for changes and open TD issues.

Lastly, it will be possible to use the quantified TD to further study how it can be used as a proactive measure, how it can impact TD management, and how it relates to other data-driven approaches that could be useful for timely resolutions, decisions, and communication.

References

[1] Khetam Al Sharou, Zhenhao Li, and Lucia Specia. “Towards a Better Understanding of Noise in Natural Language Processing”. In: *Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2021)*. 2021, pp. 53–62.

[2] Douglas G Altman and J Martin Bland. “How to obtain the P value from a confidence interval”. In: *Bmj* 343 (2011).

[3] Areti Ampatzoglou et al. “The financial aspect of managing technical debt: A systematic literature review”. In: *Information and Software Technology* 64 (2015), pp. 52–73.

[4] Paris Avgeriou et al. “Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)”. In: *Dagstuhl Reports* 6 (Jan. 2016). DOI: `10.4230/DagRep.6.4.110`.

[5] Paris Avgeriou et al. “Managing technical debt in software engineering (dagstuhl seminar 16162)”. In: *Dagstuhl Reports*. Vol. 6. 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.

[6] Paris C Avgeriou et al. “An overview and comparison of technical debt measurement tools”. In: *IEEE Software* (2020).

[7] Gabriele Bavota and Barbara Russo. “A large-scale empirical study on self-admitted technical debt”. In: *Proceedings of the 13th international conference on mining software repositories*. 2016, pp. 315–326.

[8] Stephany Bellomo et al. “Got Technical Debt? Surfacing Elusive Technical Debt in Issue Trackers”. In: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 2016, pp. 327–338.

[9] Stephany Bellomo et al. “Got technical debt? surfacing elusive technical debt in issue trackers”. In: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2016, pp. 327–338.

[10] Daniel Berrar. “Bayes’ theorem and naive Bayes classifier”. In: *Encyclopedia of Bioinformatics and Computational Biology: ABC of Bioinformatics* 403 (2018).

[11] KR1442 Chowdhary. “Natural language processing”. In: *Fundamentals of artificial intelligence* (2020), pp. 603–649.

[12] Peter J Denning. “What is software quality?” In: *Communications of the ACM* 35.1 (1992), pp. 13–15.

[13] Christof Ebert et al. “DevOps”. In: *IEEE Software* 33.3 (2016), pp. 94–100.

[14] Sergey Edunov et al. “Understanding back-translation at scale”. In: *arXiv preprint arXiv:1808.09381* (2018).

[15] Nicole Forsgren and Mik Kersten. “DevOps metrics”. In: *Communications of the ACM* 61.4 (2018), pp. 44–48.

[16] Nicole Forsgren et al. “2019 accelerate state of devops report”. In: (2019).

[17] Kavita Ganesan and Michael Subotin. “A general supervised approach to segmentation of clinical texts”. In: *2014 IEEE International Conference on Big Data (Big Data)*. IEEE. 2014, pp. 33–40.

[18] Melissa J Goertzen. “Introduction to quantitative research and data”. In: *Library Technology Reports* 53.4 (2017), pp. 12–18.

[19] Satish Gojare, Rahul Joshi, and Dhanashree Gaigaware. “Analysis and design of selenium webdriver automation testing framework”. In: *Procedia Computer Science* 50 (2015), pp. 341–346.

[20] Johanna Gustafsson. *Single case studies vs. multiple case studies: A comparative study*. 2017.

[21] Christine Howes et al. “Proceedings of the Probability and Meaning Conference (PaM 2020)”. In: *Proceedings of the Probability and Meaning Conference (PaM 2020)*. 2020.

[22] Sang-Bum Kim et al. “Some effective techniques for naive bayes text classification”. In: *IEEE transactions on knowledge and data engineering* 18.11 (2006), pp. 1457–1466.

[23] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. “Technical debt: From metaphor to theory and practice”. In: *IEEE Software* 29.6 (2012), pp. 18–21.

[24] Yu Beng Leau et al. “Software development life cycle AGILE vs traditional approaches”. In: *International Conference on Information and Network Technology*. Vol. 37. 1. 2012, pp. 162–167.

[25] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. “The Technical Debt Dataset”. In: *15th Conference on Predictive Models and Data Analytics in Software Engineering*. Jan. 2019.

[26] Valentina Lenarduzzi et al. “A systematic literature review on technical debt prioritization: strategies, processes, factors, and tools”. In: (Apr. 2019).

[27] Valentina Lenarduzzi et al. *On the Fault Proneness of SonarQube Technical Debt Violations: A comparison of eight Machine Learning Techniques*. June 2019.

[28] Valentina Lenarduzzi et al. “Technical Debt Impacting Lead-Times: An Exploratory Study”. In: *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2021, pp. 188–195.

[29] Valentina Lenarduzzi et al. “Towards Surgically-Precise Technical Debt Estimation: Early Results and Research Roadmap”. In: Aug. 2019. ISBN: 978-1-4503-6855-1. DOI: `10.1145/3340482.3342747`.

[30] Zengyang Li, Paris Avgeriou, and Peng Liang. “A Systematic Mapping Study on Technical Debt and Its Management”. In: *Journal of Systems and Software* (Dec. 2014). DOI: `10.1016/j.jss.2014.12.027`.

[31] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. “Recurrent neural network for text classification with multi-task learning”. In: *arXiv preprint arXiv:1605.05101* (2016).

[32] Zhongxin Liu et al. “Satd detector: A text-mining-based self-admitted technical debt detection tool”. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 2018, pp. 9–12.

[33] Edward Loper and Steven Bird. “Nltk: The natural language toolkit”. In: *arXiv preprint cs/0205028* (2002).

[34] Antonio Martini. “Anacondebt: a tool to assess and track technical debt”. In: *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE. 2018, pp. 55–56.

[35] Antonio Martini, Viktoria Stray, and Nils Brede Moe. “Technical-, social-and process debt in large-scale agile: an exploratory case-study”. In: *International Conference on Agile Software Development*. Springer. 2019, pp. 112–119.

[36] Wes McKinney et al. “pandas: a foundational Python library for data analysis and statistics”. In: *Python for high performance and scientific computing* 14.9 (2011), pp. 1–9.

[37] Ipek Ozkaya, RL Nord Z Kurtz, and RS Sangwan. *Automatically Detecting Technical Debt Discussions*. Tech. rep. Technical Report. Carnegie Mellon University: Software Engineering Institute, 2019.

[38] Cristian Padurariu and Mihaela Elena Breaban. “Dealing with data imbalance in text classification”. In: *Procedia Computer Science* 159 (2019), pp. 736–745.

[39] Fabian Pedregosa et al. “Scikit-learn: Machine learning in Python”. In: *the Journal of machine Learning research* 12 (2011), pp. 2825–2830.

[40] Maria Pelevina et al. “Making sense of word embeddings”. In: *arXiv preprint arXiv:1708.03390* (2017).

[41] Jeffrey Pennington, Richard Socher, and Christopher D Manning. “Glove: Global vectors for word representation”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[42] Aniket Potdar and Emad Shihab. “An Exploratory Study on Self-Admitted Technical Debt”. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014, pp. 91–100. DOI: `10.1109/ICSME.2014.31`.

[43] Vanshika Rastogi. "Software development life cycle models-comparison, consequences". In: *International Journal of Computer Science and Information Technologies* 6.1 (2015), pp. 168–172.

[44] Nayan B. Ruparelia. "Software Development Lifecycle Models". In: *SIGSOFT Softw. Eng. Notes* 35.3 (May 2010), pp. 8–13. ISSN: 0163-5948. DOI: 10.1145/1764810.1764814. URL: https://doi.org/10.1145/1764810.1764814.

[45] Nyyti Saarimaki et al. "On the accuracy of sonarqube technical debt remediation time". In: *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2019, pp. 317–324.

[46] Mary Sánchez-Gordón and Ricardo Colomo-Palacios. "Characterizing DevOps culture: a systematic literature review". In: *International Conference on Software Process Improvement and Capability Determination*. Springer. 2018, pp. 3–15.

[47] Walt Scacchi et al. *Understanding free/open source software development processes*. 2006.

[48] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. "Using natural language processing to automatically detect self-admitted technical debt". In: *IEEE Transactions on Software Engineering* 43.11 (2017), pp. 1044–1062.

[49] Daricélio Moreira Soares et al. "Acceptance factors of pull requests in open-source projects". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. 2015, pp. 1541–1546.

[50] Edith Tom, Aybüke Aurum, and Richard Vidgen. "An exploration of technical debt". In: *Journal of Systems and Software* 86.6 (2013), pp. 1498–1516.

[51] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation". In: *Computing in science & engineering* 13.2 (2011), pp. 22–30.

[52] Bo Zhao. "Web scraping". In: *Encyclopedia of big data* (2017), pp. 1–3.

Appendix A

External materials

The raw data, scripts that have been used, and final data from this thesis have been published in a Github repository for replication purposes. This can be viewed on the URL:

`https://github.com/Danielskry/IN5960-Master-Thesis`.