

Automatic priority classification for issues in issue tracking systems using transformers

*A design science research approach to
classification of priority of issues in issue
tracking systems*

Kristian Marison Haugerud



Thesis submitted for the degree of
Master in in Informatics: Programming and system
architecture
60 credits

Institute for Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2024

Automatic priority classification for issues in issue tracking systems using transformers

*A design science research approach to
classification of priority of issues in
issue tracking systems*

Kristian Marison Haugerud

© 2024 Kristian Marison Haugerud

Automatic priority classification for issues in issue tracking systems using
transformers

<http://www.duo.uio.no/>

Printed: Reprosentralen, University of Oslo

Abstract

This thesis presents a design science research approach to automatic classification of issue priority in issue tracking systems using transformer models. The study addresses the need for efficient prioritization of issues in software development, where accurate and timely classification can significantly impact project results.

The methods used include transformer models such as DistilBERT and DeBERTaV3, evaluated through multiple iterations. The first iteration explored multilabel classification, while subsequent iterations focused on binary classification and improving data quality. Data preprocessing and cleaning were performed on both GitHub and Jira datasets to ensure the reliability of the models.

The models were first tested on a holdout test set, which showed particularly good results for the GitHub model. Then, they were tested on projects unseen by the model. Here, the results were more mixed, with some projects performing reasonably well while others were almost as good as random guesses. The final artifact demonstrated improved accuracy and reliability in real-world scenarios when fine-tuned on specific projects.

In a best-case scenario, after fine-tuning, the best results were observed with the Jira model tested on the GitHub project Flutter. It correctly labeled around 80% high-priority and non-high-priority issues. However, the models did not achieve these results for all projects unseen by the model. The study concludes that while transformer models hold promise for automating the classification of issue priority, more research is necessary to enhance their generalizability and effectiveness across various projects.

Preface

I would like to express my sincere gratitude to my main supervisor, Antonio Martini, for their continuous support, patience, and invaluable guidance. Their insightful feedback and encouragement have been instrumental in shaping this thesis.

I am also grateful to my co-supervisor, Karthik Sankar, for his expertise and constructive advice. Their contributions have significantly enhanced the quality of this work.

Special thanks go to the *Machine learning infrastructure (ML Nodes), University Centre for Information Technology, University Of Oslo, Norway* for providing the necessary resources and a conducive environment for research.

Finally, I would like to thank everyone who contributed to this thesis, directly or indirectly.

Contents

1	Introduction	1
1.1	Research questions	2
1.2	Structure	3
1.2.1	Chapter 2: Background	3
1.2.2	Chapter 3: Methodology	3
1.2.3	Chapter 4: Iterations	3
1.2.4	Chapter 5: Final Artifact	3
1.2.5	Chapter 6: Discussion	4
1.2.6	Chapter 7: Conclusion	4
2	Background	5
2.1	Related work	5
2.1.1	Prioritizing tasks in software development: A systematic literature review	5
2.1.2	Understanding and Enhancing Issue Prioritization in GitHub	8
2.1.3	Predicting the Objective and Priority of Issue Reports in Software Repositories	9
2.1.4	Toward an Effective Bug Triage System Using Transformers to Add New Developers	10
2.1.5	Findings from the studies	11
2.2	Machine Learning and Natural Language Processing Techniques	12
2.2.1	Introduction to Natural Language Processing	12
2.2.2	Machine Learning in NLP	12
2.2.3	Transformers for text classifications	12
2.3	Issue tracking systems	14
2.3.1	Jira	14
2.3.2	GitHub issue tracking system	15
2.3.3	Lack of Labels of in GitHub projects	16
3	Methodology	18
3.1	Design Science Research	18
3.1.1	DSR process	18
3.1.2	Iterations	20
3.2	Activity 1: Problem identification and motivation	20

3.2.1	Identifying the problem for the thesis.	21
3.2.2	Gap in in the research	21
3.2.3	Implication of solution	21
3.2.4	Stakeholder of the final artifact	22
3.2.5	Practical Use Case	22
3.3	Activity 2: Define the Objectives of a Solution	23
3.3.1	Objectives of the Thesis	23
3.4	Activity 3: Design and development	25
3.5	Activity 4: Demonstration	25
3.6	Activity 5: Evaluation	25
3.6.1	Evaluating the Models	25
3.6.2	Activity 6: Communication	31
3.7	Design and development of the final artifact	31
3.8	GitHub data set	32
3.8.1	Dataset creation and extraction of labels	32
3.8.2	Preprocessing	33
3.9	Jira public dataset	34
3.9.1	Data set creation	35
3.9.2	Jira dataset after cleaning	36
3.9.3	Including the Jira data Set	37
3.9.4	High, medium, low, lowest vs highest priority	38
3.9.5	High, medium, low, lowest vs Highest min length 25th percentile	39
3.9.6	Medium, low, lowest vs highest, high	39
3.9.7	Lowest vs Highest priority	41
3.9.8	Low, lowest vs highest, high priority	41
3.10	Preprocessing and data cleaning of both Jira and GitHub data sets	42
3.11	Models used and model selection	47
3.11.1	DistilBERT	47
3.11.2	DeBERTaV3	47
4	Iterations	49
4.1	Design and development for the iterations.	49
4.2	Iteration 1. Multi-label classification. High vs. medium vs. low priority	50
4.2.1	Objectives	50
4.2.2	Design and development	50
4.2.3	Evaluation	50
4.3	Iteration 2: Binary classifications.	51
4.3.1	Objectives	51
4.3.2	Design and development	51
4.3.3	Evaluation	52
4.3.4	Why use binary over multi-label text classification . .	53
4.4	Iteration 3: Improve the quality of the data set	54
4.4.1	Objectives	54
4.4.2	Design and Development	54
4.4.3	Evaluation	56

5	Final Artifact	58
5.1	Jira: medium, low, lowest vs. highest and High	58
5.2	Jira: lowest vs. highest	59
5.3	Jira: low, lowest vs. highest, high	60
5.4	Summary of results for RQ1.1	61
5.4.1	GitHub models	61
5.4.2	Jira models	62
5.5	Jira: high, medium, low, lowest vs. Highest priority	63
5.5.1	RQ1.1: Training and test on holdout test set	63
5.5.2	RQ1.2 and RQ1.3: Inference and fine-tuning on projects unseen by the model	64
5.6	GitHub: medium and low vs. high priority. Top50	67
5.6.1	RQ1.1: Training and inference in the holdout test set	67
5.6.2	RQ1.2: Inference on unseen projects	67
5.6.3	RQ1.3: Fine-tuning on unseen projects	68
6	Discussion	70
6.1	Discussion of results of the research questions	70
6.1.1	Recall of classified high priority issues	70
6.1.2	The precision of classified high priority issues, how many were mistakenly classified?	70
6.1.3	RQ1.1: How well do Transformers perform for priority classification when the model is trained and tested on the same dataset?	71
6.1.4	RQ1.2: How well do Transformers generalize to classify priority in projects unseen by the model?	74
6.1.5	RQ1.3: How well do Transformers perform for priority classification when the model is fine-tuned?	76
6.1.6	RQ1: How well do Transformers support automatic priority classification of issues in issue tracking systems?	79
6.2	Related work	80
6.3	Contributions	81
6.4	Implications	81
6.4.1	Implications for research	81
6.4.2	Implications for practice	82
6.5	Quality of research and limitations	84
6.5.1	Validity and Validation	84
6.5.2	Construct validity threat	84
6.5.3	Internal validity	85
6.5.4	External validity	85
6.5.5	Conclusion validity	86
6.5.6	Limitations	87
6.6	Final thoughts	87
6.7	Future work	88
7	Conclusion	89

8	Appendix	95
8.1	Code	95
8.2	Use of AI	95

List of Figures

2.1	Default priority levels and description suggested from Jira Documentation [30]	15
2.2	Screenshot: Example of a GitHub issues from the rust-lang/rust repository.	15
2.3	New issues created in one month in the vscode repository. .	16
2.4	Number of unique users creating issues in the vscode repository.	16
3.1	DSR Methodology Process model. Adopted from Brocke et al. [4]. Showing each activity in the methodology	19
3.2	The different iterations leading to the final artifact	20
3.3	Formula for Accuracy	29
3.4	Formula for Precision	29
3.5	Formula for Recall	29
3.6	Formula for the F1 Score	29
3.7	Formula for the Matthews Correlation Coefficient	30
3.8	This shows the design and development workflow and how the outcomes are related to the research questions.	32
3.9	Jira data set creation workflow each results in data set used in the final artifact	36
3.10	Jira: high, medium, low, lowest vs. highest priority with minimum length of the 25th percentile: Distribution of issues length	40
3.11	Preprocessing and data cleaning process.	43
3.12	Example of bot created issue in AdguardFilters GitHub repository ¹	44
3.13	Example of a description of an issue created by a bot in AdguardFilters GitHub repository ²	45
6.1	Confusion matrix using the Jira model with inference on the flutter project.	78
6.2	Confusion matrix after fine-tuning the Jira model with the flutter project.	79

List of Tables

2.1	Different methods used to prioritize tasks found by [5] . . .	6
2.2	Number of studies on priority of tasks found by Bugayenko et al. [5]	6
2.3	Number of different data sets used in prioritizing task found by Bugayenko et al. [5]. Note: A paper could use several data sets	7
2.4	Results from Izadi et al. [14] looking at low priority vs. high priority using feature engineering.	10
2.5	Definition of priority labels used in the flutter/flutter repository and count per labels as of 25.09.2023.	16
3.1	Quadrants of a confusion Matrix. True positive and true negative are the correct answers and the labels we want [7] .	30
3.2	Explanation of Terms in the Context of Priority Issue Classification	31
3.3	Overview number of issues in the GitHub data set.	33
3.4	The number of issues in each priority level and descriptions	34
3.5	Number of issues per class after cleaning in GitHub data set.	34
3.6	MongoDB priority levels mapped to a common set of priority levels.	35
3.7	Number of issues per class in Jira data set used to create training data sets after cleaning.	37
3.8	Summary of dataset token metrics and issue counts in Jira data set.	37
3.9	Jira: high, medium, low, lowest vs highest class distribution.	38
3.10	Issue lengths summary Statistics before and after removing the lowest 25th percentile.	40
3.11	Jira: high, medium, low, lowest vs highest 25th minimum length class distribution	40
3.12	Jira: medium, low, lowest vs highest, high class Distribution	41
3.13	Jira: lowest vs. highest priority class Distribution	41
3.14	Jira low, lowest vs highest, high priority: Class Distribution .	42
4.1	Number of issues per class in high vs. medium vs. low. . . .	50
4.2	Results of Multi label classification on hold out test set. . . .	51
4.3	Result from binary classifications.	53

4.4	Number of issues before and after removing issues with fewer than 100 characters	55
4.5	Number of issues before and after removing issues with fewer than 500 characters	55
4.6	Number of issues before and after removing issues with fewer than 1000 characters	56
4.7	Results from models trained with DistilBERT after improving quality of the data.	57
5.1	Jira: medium, low, lowest vs. highest, high inference on holdout test set	59
5.2	Jira: lowest vs. highest. Inference on holdout test set and projects unseen by the model.	60
5.3	Jira: Low, Lowest vs Highest, High. Inference on test sets	61
5.4	Overview of all the different models trained with DeBERTaV3	63
5.5	Jira: high, medium, low, lowest vs. highest dataset Distribution	64
5.6	Testing on hold out test During Inference Rest vs Highest priority	64
5.7	Jira: high, medium, low and lowest vs. highest priority. Inference on projects unseen by the model and fine tuning on them with 30% of data from the project included in the training data set.	66
5.8	GitHub: med, low vs. high Dataset Distribution	67
5.9	GitHub: medium and low vs. high priority. Inference on the hold out test set.	67
5.10	GitHub: medium and low vs high priority. Inference on projects unseen by the model and fine tuning on them with 30% of data from the project included in the training data set.	69
6.1	Comparing results from this thesis with results from Izadi et al. [14]	80

Chapter 1

Introduction

In software engineering, prioritizing tasks is of significant importance [17]. Bugayenko et al. [5] highlight that task prioritization plays a pivotal role in efficient resource allocation, allowing teams to address high-priority issues.

Through prioritization, teams can focus their efforts on resolving the most important issues that have the potential to influence the project. Priority issues help teams meet deadlines and ensure that the most important tasks are completed. In addition, prioritization contributes to improved overall project management and team productivity by delineating a clear roadmap of tasks that require attention.

The assignment of priority labels in GitHub projects is underutilized, as He et al. [12] has shown. This also aligns with my own findings, where I look into the use of priority issues in the *vscode* project on GitHub, the findings are presented in Section 2.3.3. A model to automatically label issues with a priority would help developers deal with the increasing number of new issues created daily.

In this thesis, the focus is on using the state-of-the-art machine learning model Transformers to assign priority to issues in the GitHub and Jira issue tracking systems. An issue is a form of task that describes things like a bug or a new feature that developers have to deal with.

The motivation for this research came from the results of Skryseth et al. [25]. Their work involved training Transformer models on GitHub issues to detect technical debt, which yielded promising results. This inspired the exploration of automating the prioritization of technical debt. However, the number of technical debt and priority issues was too small to train a transformer. The scope of the thesis expanded to include the prioritization of all types of issues, including bugs, feature requests, and improvements.

The main goal of the thesis is to automatically prioritize issues, with an emphasis on high-priority issues. The thesis will explore how effective this

approach is by evaluating the performance of the models.

If the results are good, the model could be integrated into a tool. Such as a GitHub marketplace application, or integrating it into Jira. This would allow developers to receive suggestions on priority labels for newly created issues. In addition, it could be utilized for a backlog of issues that currently do not have a priority label. These objectives led to the following research questions.

1.1 Research questions

RQ1: How well do Transformers support automatic priority classification of issues in issue tracking systems?

Transformers have emerged as powerful models for natural language processing tasks, demonstrating their ability to effectively understand and process textual data. In the context of issue tracking systems, this research question is crucial. This study evaluates the effectiveness of Transformer models in categorizing issues based on their priority levels. By exploring this research question, insights into the optimal approach for issue prioritization will be explored. Whether it is through binary or multi-label classification methods.

To help find the best model, several sub-research questions need to be answered. They help to find how well the models generalize to new unseen projects and their adaptability when fine-tuned on a project.

RQ1.1: How well do Transformers perform for priority classification when the model is trained and tested on the same dataset?

When training a machine learning model, the dataset is typically divided into training, validation, and test sets. The training and validation sets are used to train the model, while the test set is reserved for evaluating the model's performance. Since the test set has not been seen by the model during training, it provides an unbiased assessment of the model's effectiveness. Answering this research question will help identify the most suitable model or approach for the task at hand. Although this analysis offers a foundational understanding of the model's performance, it does not indicate how well it generalizes to projects unseen by the model.

RQ1.2: How well do Transformers generalize to classify priority in projects unseen by the model?

This sub-question aims to investigate the generalization ability of Transformers in classifying the priority of issues in new projects. Several test projects are withheld from the model during training. This will demonstrate how well the models perform on projects unseen by the models and provide insight into the practical effectiveness of the models. In addition,

this research question serves as a key metric for evaluating project performance, highlighting the overall effectiveness of the model. This is crucial to ensure that the deployment of such models in practical settings is reliable and beneficial across various software projects and technical environments.

RQ1.3: How well do Transformers perform for priority classification when the model is fine-tuned?

When fine-tuning, data from unseen projects are introduced to the model. Here, the same project from Section 1.1 is used and comparisons are made to assess improvements from fine-tuning. This process aims to evaluate the model's performance when specifically trained on project-specific or dataset-specific data, demonstrating its adaptability to new information. If the results are promising, this approach could be an effective strategy for training the model with data relevant to the projects where it will be applied.

1.2 Structure

This section provides an overview of the different chapters.

1.2.1 Chapter 2: Background

The background section addresses the thesis's most important topics: Prioritization of software items, issue tracking systems, and transformers for text classification. It also covers related work and summarizes the research conducted on the topic to date.

1.2.2 Chapter 3: Methodology

This chapter presents the research methodology, design science research. It provides an overview of the various research strategies and methods used.

1.2.3 Chapter 4: Iterations

Here the iterations that lead to the final artifact are presented. The results here are partial results that help to explain why the choices were made for the final artifact.

1.2.4 Chapter 5: Final Artifact

The results of the final artifact addressing the research questions are presented.

1.2.5 Chapter 6: Discussion

This chapter reflects on the research and the findings of the thesis related to the research questions. The chapter discusses the contributions, limitations, quality of research and practical and theoretical implications and suggests areas for future work.

1.2.6 Chapter 7: Conclusion

This chapter concludes the work and presents recommendations for future research.

Chapter 2

Background

This chapter provides an overview of the relevant concepts of the thesis. The aim is to provide a contextual foundation for understanding the project. First, the related work and their most important findings are presented. Then, the theory about machine learning, natural language processing, and transformers is presented to give an understanding of why these tools have been used. Lastly, issues and issue tracking systems are explained. The differences and commonalities between Jira and GitHub are also presented.

2.1 Related work

In this section the related work and their most important findings related to the thesis presented are presented.

2.1.1 Prioritizing tasks in software development: A systematic literature review

The most recent systematic literature review (SLR) on prioritization of tasks in software development was carried out by Bugayenko et al. [5]. The review sheds light on the evolution and current status of task prioritization within software development. It emphasizes that bug prioritization has historically been the primary focus. Recent research has expanded to include pull request and issue prioritization, driven by the widespread adoption of version control and issue management systems, including GitHub and Jira. In particular, in issue prioritization, there is a growing interest in understanding how issues are chosen and ranked within software projects, mirroring the increasing complexity and scale of modern development environments. The review also pinpoints the metrics most commonly used to evaluate prioritization models, such as the F score, precision, recall, and accuracy, highlighting their crucial role in evaluating the effectiveness of the model.

The review highlights a significant shift from conventional techniques

to machine learning-based approaches, mirroring the larger trends in software engineering research. This shift indicates a possible increase in the use of advanced analytical techniques that can manage complex and ever-changing prioritization scenarios.

Furthermore, the study offers a critical assessment of the array of methods and models used in various research works, laying the groundwork for future studies focused on improving task prioritization processes within agile software development environments. This analysis not only aids in identifying the most effective strategies but also highlights gaps and potential areas for further investigation, particularly in the integration of machine learning techniques into real-world applications.

The SLR also identifies the common algorithms listed in Table 2.1. However, the results indicate that none of the studies utilize transformers for prediction purposes. This underscores the importance of further exploration into the application of transformers to predict the priorities of the issues.

Method	Count
Naive Bayes	31
Support Vector Machine	29
K-Nearest Neighbours	23
Neural Network	14
Random Forest	20
Multinomial Naive Bayes	18
Decision tree	8
Linear Regression	10
J48	7
DRONE	4

Table 2.1: Different methods used to prioritize tasks found by [5]

The SLR highlights that one of the studies done on priority in software development is done on bug prioritization. Bugayenko et al. [5] only found four studies that focused only on the priority of the issues as seen in Table 2.2. Highlighting the need to explore assigning priority to all issues and not only bugs.

Topic	Earliest date	Latest date	Quantity
Bug prioritization	2010	2022	31
Bug severity prediction	2012	2022	43
Issue priority	2014	2020	4
Pull Request priority	2014	2021	6

Table 2.2: Number of studies on priority of tasks found by Bugayenko et al. [5]

They also list the different data sets used by the paper, and it shows that a GitHub data set is 11 of the total 113 data sets that were used to train a

machine learning model. Highlighting the need for explorations of other datasets such as Jira and GitHub.

Platform	Number of Issues
Eclipse	45
Mozilla	27
Firefox	9
GitHub	11
jdt	5
cdt	6
pde	4
GNOME	6

Table 2.3: Number of different data sets used in prioritizing task found by Bugayenko et al. [5]. Note: A paper could use several data sets

Issue prioritization

The SLR highlights the work by Zaidi et al. [33] as the most promising in the field of issue prioritization. The study addresses the challenges of managing a large volume of issues on GitHub using a machine learning-based tool known as the Issue Prioritizer.

This tool evaluates issues based on three main criteria: the lifetime of the issue, its 'hotness', indication of recent activity, and the category to which the issue belongs. The architecture of the tool comprises a Watcher module for issue tracking, an Analyzer for data extraction, a Predictor that employs machine learning models to predict issue resolution time, and a Visualizer for displaying prioritized issues.

The study emphasizes the potential of the Issue Prioritizer in assisting developers to efficiently manage and prioritize GitHub issues, thereby enhancing their ability to address the influx of both new and ongoing issues in large projects.

Machine learning models were used to predict the lifetime of an issue based on various features, such as the number of assignees, comments, and author association. The Decision Tree and Random Forest classifiers emerged as the best performing models, both achieving an accuracy of 48%, suggesting moderate effectiveness in predicting issue resolution times.

The results indicate that while Issue Prioritizer effectively utilizes activity-based metrics to assess issue importance, further enhancements and broader testing are essential. Specifically, improvements in lifetime prediction models and the expansion of the evaluation to encompass a wider range of GitHub projects are necessary for more generalizable outcomes.

2.1.2 Understanding and Enhancing Issue Prioritization in GitHub

The paper by He et al. [12] provides a comprehensive analysis of issue prioritization mechanisms on GitHub, particularly the role of labels and other issue features in prioritizing tasks within software projects. The empirical study examines a dataset comprising more than 1.5 million issues from 300 different GitHub projects to understand and improve the prioritization process. Despite the widespread use of labels to categorize and prioritize issues, their actual effectiveness in influencing issue handlers' prioritization decisions is limited. Key findings indicate that only about half of the issues in GitHub projects are labeled, and those with labels do not necessarily receive higher priority. This is surprising, as labels are intended to facilitate the prioritization process by marking issues according to urgency and type.

The authors also conducted a thorough examination of the role of labels in issue prioritization on GitHub and analyzed prioritization patterns based on various issue features.

The research revealed that issues without labels tend to receive a higher priority than those with labels, consistently observed across projects of different sizes. Specifically, issues without labels had higher median and mean priority scores compared to labeled issues, a statistically significant finding. Issues labeled with high priority labels did not necessarily receive faster attention compared to those labeled low priority or those without priority-specific labels, suggesting a gap between the intended and actual impact of labels.

The study used a Generalized Linear Mixed Model to analyze how different features of issues influence their prioritization. Significant characteristics affecting the priority of the issue included the presence of an assignee, the length of the title and body of the issue, and the specific types of labels. Interestingly, factors like the number of comments or the issue's age were found to be less influential, suggesting the presence of other subtle factors in issue prioritization.

The paper explored the effectiveness of various learning-to-rank algorithms in prioritizing issues based on the significant features identified. Algorithms such as AdaRank and random forest-based methods outperformed others in ranking issues effectively according to calculated priorities, indicating the potential of advanced machine learning techniques in improving issue prioritization processes on platforms like GitHub.

The study sheds light on the practical challenges of issue management on major software development platforms such as GitHub, emphasizing the limitations of current labeling practices and the potential for data-driven algorithmic approaches to enhance issue prioritization in large-scale software development environments.

2.1.3 Predicting the Objective and Priority of Issue Reports in Software Repositories

Do et al. [9] is the closest paper found that has a similar approach to prioritize issues in issues tracking systems, but does not train the model directly on the raw text of the description. They are using high- and low-priority issues from open-source projects on GitHub.

The authors used an approach that integrates feature engineering, machine learning, and textual analysis to prioritize issues in the GitHub software repositories. They collected and pre-processed GitHub issue data, including titles, descriptions, and labels. The key steps involved extensive feature engineering to transform the raw data into a structured form suitable for machine learning models. This included creating features from textual information through TF-IDF vectorization and sentiment analysis, together with extracting discussion-related, event-related, developer-related, and sentiment features from issue metadata.

The engineered features were normalized and balanced to address data scale disparities and class imbalance. A Random Forest classifier was trained to predict high- or low-priority levels of issues. This model was evaluated in project-specific and cross-project contexts to assess its effectiveness and generalizability. To validate the model predictions on unseen data, a human labeling and evaluation task was conducted with software engineers, further confirming the model's accuracy.

This approach encapsulates a comprehensive strategy for issue prioritization, combining advanced data processing techniques with machine learning to automate and enhance the management of issue reports in software development projects.

In the priority prediction task, the study focused on a subset of the initial data set specifically tailored to evaluate and predict the priority levels of the issue reports. This subset is characterized by the following metrics:

- Total Issues for Priority Prediction: The dataset for this specific task included 82,719 issues, strategically chosen to represent a balanced view of priority levels in various projects.
- Number of repositories analyzed: The issues were sourced from 70 distinct repositories, which were selected based on their use of Java as the main programming language and the presence of sufficiently labeled issue reports for training and testing.

Classification of Priority Levels:

- High Priority (HP): This class consisted of 44,733 issues identified as critical or urgent, which require swift action to resolve. These issues often have a significant impact on the project's functionality or user experience.
- Low Priority (LP): Comprising 37,986 issues, this category includes

reports that, while still important, do not require immediate attention and can be scheduled for resolution according to standard maintenance practices.

This data ensures a robust framework for developing and validating the efficacy of machine learning models in automating the issue prioritization process, catering to the nuanced needs of software development and maintenance.

Their approach outperformed other methods with an accuracy of 74% as seen in Table 2.4. The model performed worse in high priority, especially in recall, with 59% in HP vs. 83% in LP.

Dataset	Precision	Recall	F1-score	Accuracy
Project-based priority prediction results				
Low Priority	0.77	0.78	0.77	0.75
High Priority	0.73	0.74	0.72	
Cross-project priority prediction results				
Low Priority	0.75	0.83	0.79	0.74
High Priority	0.70	0.59	0.64	

Table 2.4: Results from Izadi et al. [14] looking at low priority vs. high priority using feature engineering.

2.1.4 Toward an Effective Bug Triage System Using Transformers to Add New Developers

In their study titled Toward an Effective Bug Triage System Using Transformers to Add New Developers, Zaidi et al. [33] makes a significant advancement in automated bug triage systems. They address the complexities of bug assignment, especially in integrating new developers into established triage systems without requiring a full model retraining. The research introduces a unique application of the Bidirectional Encoder Representations from Transformers model, utilizing its deep learning capabilities to understand bug reports on a semantic level.

The methodology focuses on preprocessing and tokenization of issue descriptions and summaries from large-scale open source projects such as Eclipse and Mozilla, creating a model that is finely tuned to software bug reports.

A key innovation is the fine-tuning process to include new developers based on a simulated triage history, enabling a smooth transition without the need for full model retraining. This is achieved by assuming a minimal set of triage activities for new developers, which is then used to adjust the existing model, demonstrating a flexible and efficient approach to updating bug triage systems in dynamic software development environments.

The effectiveness of this approach is assessed through top-k accuracy metrics, showing that it performs more accurately predicting bug assignments

compared to traditional embedding techniques such as Word2Vec, GloVe, and ELMo. This study not only showcases the potential of transformer-based models in improving bug triage systems' precision but also tackles the practical challenge of integrating new developers into these systems, making a significant contribution to automated software maintenance and development.

This approach is more similar to the thesis approach in that it looks at the description of the issues to train the model and uses transformers. However, Zaidi et al. [33] are just focusing on issues labeled bug. In this thesis, all types of labels are considered.

2.1.5 Findings from the studies

He et al. [12] found that the number of comments and the age of an issue are less influential in the prioritization process. This finding suggests that looking at the title and descriptions of the issue is worth exploring for prioritization. They also found that using ML models to assign priority to issues can be a viable method. However, they did not explore the use of state-of-the-art machine learning models using transformers, highlighting the need to investigate this type of machine learning model for analyzing issue descriptions.

Do et al. [9] focused on differentiating between high-priority and low-priority issues by examining their features. Their approach did not incorporate transformers, but still provided valuable insights into issue prioritization. On the other hand, Zaidi et al. [33] used transformers to analyze issue descriptions, but their work was focused on prioritizing bugs. This indicates a gap in research where transformers are used for a broader range of issues beyond just bugs.

A systematic literature review from 2023 by Bugayenko et al. [5] did not find any papers using transformers to prioritize issues, strengthening the novelty of exploring transformers for this purpose. Skryseth et al. [25] showed promising results using transformers to identify technical debt in issues, suggesting the potential effectiveness of transformers in this domain.

Based on these findings, it is evident that no other papers have been written on classification of priority in issues using both the descriptions text and state-of-the-art transformers comprehensively. Furthermore, none of the studies reviewed used the Public Jira dataset [20], although some used the GitHub datasets.

This thesis aims to fill this gap by using a more extensive dataset collected by Skryseth et al. [25] and the Jira data set. Then, transformers are applied to analyze and prioritize a wide range of issues, including bugs, feature requests, and improvements. This approach has the potential to improve the automation and accuracy of issue prioritization in issue tracking systems, Jira, and GitHub.

2.2 Machine Learning and Natural Language Processing Techniques

2.2.1 Introduction to Natural Language Processing

Natural Language Processing (NLP) is a branch of artificial intelligence that focuses on the interaction between computers and humans using natural language. The primary objective of NLP is to enable computers to comprehend, interpret, and manipulate human language in a way that is both meaningful and practical. In its early stages, the field relied on rule-based methods, which were later replaced by statistical approaches in the 1990s with the rise of machine learning techniques [16]. The evolution of machine learning models has greatly improved the capabilities of NLP systems, enabling more advanced and automated processing of extensive datasets.

2.2.2 Machine Learning in NLP

Machine learning in NLP involves training a model on a dataset so that it can perform a specific task, such as text classification, sentiment analysis, or language translation. Machine learning models for NLP are generally divided into supervised and unsupervised learning paradigms. Supervised learning models, including support vector machines and neural networks, require a labeled dataset and focus on classification or regression tasks [1]. Unsupervised learning models, such as clustering and dimensionality reduction techniques, explore patterns in data without pre-assigned labels [10].

Machine learning, especially in the context of NLP, has undergone significant transformations. Traditional machine learning methods for text classification, such as Naive Bayes, Decision Trees, and Support Vector Machines, rely heavily on handcrafted features and substantial domain expertise [1]. However, there has been a shift in the field with the emergence of neural network-based approaches. These approaches allow systems to learn features directly from the data, eliminating the need for explicit programming for each task [13].

2.2.3 Transformers for text classifications

The development of the transformer model by Vaswani et al. [29] marked a revolutionary advance in deep learning for NLP. Unlike previous models that processed data sequentially (e.g., recurrent neural networks), transformers use a mechanism called self-attention to process all tokens in the input data in parallel. This architectural change significantly improves the efficiency of training models on large datasets and has led to improvements in a large number of NLP tasks, including text classification [29]. The availability of pre-trained language models developed by HuggingFace [32] contributed to the popularity and utility of Transformers. These models can be easily accessed and extended using an API.

Transformers have been the backbone of many state-of-the-art NLP models used for text classification tasks. Models like Bidirectional Encoder Representations from Transformers (BERT) and Generative Pre-trained Transformer (GPT) utilize variations of the transformer architecture and have set new benchmarks in language understanding tasks [8]. These models are pre-trained on large text datasets and fine-tuned on specific tasks, which allows them to achieve exceptional performance across diverse NLP challenges.

The primary advantage of transformers over earlier models, such as bidirectional RNNs or long-short-term memory networks, is their superior ability to capture contextual relationships between words within a text. Unlike their predecessors, which process text in a fixed sequential order, transformers view the input as a set. This allows them to assess the context of each word in relation to all other words in the sentence simultaneously, thus improving their grasp of language nuances [19]. They employ an encoder-decoder structure that processes all input data simultaneously, ensuring that the interpretation of the initial parts of a sequence directly influences the encoding of subsequent parts [6]. This self-attention mechanism significantly improves connectivity and preserves context in long sequences of tokens or text, thereby improving the overall coherence and contextual relevance of the output [29].

In this thesis, a supervised machine learning approach is employed in which labeled textual data are gathered from a variety of sources to create training datasets for machine learning algorithms. These sources range from structured online platforms and social networks to customer feedback, help requests, and survey responses.

However, extracting meaningful insights from these data presents significant challenges. The unstructured nature of many data sources, combined with the complexities of language and contextual nuances, makes this process both daunting and time consuming. With the volume of textual data growing annually and the pace of data evolution accelerating, there is an increasing need for automated text categorization techniques. Such tools are vital for businesses, helping to identify trends such as sentiment and data prioritization, as explored in this study [22].

Despite their success, transformers are not without challenges. Their large size and the need for extensive computational resources pose practical constraints and environmental concerns [26]. Moreover, issues such as model bias and interpretability remain significant challenges in the deployment of these models in real-world applications [2]. Future research is directed toward making these models more efficient, ethical and understandable.

In this thesis, DistilBERT [24] was utilized for most of the experiments. DistilBERT, being a smaller, faster, and more cost-effective variant of BERT, helped conserve both time and computational resources throughout the thesis.

2.3 Issue tracking systems

Issue tracking systems (ITS) are software tools that are used to manage and maintain a list of issues, commonly used in software development. This allows teams to record, report and track issues that need to be addressed, such as bugs, feature requests, and improvements [20]. Issues are units of information that usually include a title, description, labels, status, priority, comments, and more. Each issue can be given one or more labels such as bug, enhancement, technical debt, and priority. Other tools like a Kanban board are often included in ITSs to allow the team to organize and visualize issues.

ITS are important for both developers and management. They offer a way to communicate problems between developers and management. It offers a way to prioritize and visualize the problems so that nothing gets forgotten and the most important problem is solved first. In many cases, the ITS is publicly available, and user can submit issues with bugs and feature requests.

ITS are software tools that are commonly used in software development to manage and maintain a list of issues. These systems enable teams to record, report and track issues that require attention, such as bugs, feature requests, and improvements [20]. Issues typically consist of a title, description, labels, status, priority, comments, and more. Each issue can be assigned one or more labels, such as bug, enhancement, technical debt, and priority. In addition, ITSs often incorporate tools such as Kanban boards to help teams organize and visualize issues. ITSs play a crucial role for both developers and management by facilitating communication and enabling prioritization and visualization of problems to ensure that critical issues are addressed promptly. In many instances, ITSs are publicly accessible, allowing users to submit bug reports and feature requests.

2.3.1 Jira

Jira ¹ is a proprietary issue tracking product developed by Atlassian. It is an agile planning platform that offers more than just issue tracking with tools such as scrum boards, Kanban boards, roadmaps, reports, and more. Jira is the most popular popular tool for tracking issues and agile project management according to Datalyze², and Enlyft³.

However, Montgomery et al. [20] found that Jira is underrepresented in software engineering research. They searched Google Scholar articles with keyword GitHub and found 4,450 results and with Jira 650 results. Jira repositories can be both private and public. These repositories are public and open to anyone who wants to submit issues. Issues are a mix of issues from anyone wanting to report a bug or suggest a feature, and issues from

¹Jira <https://www.atlassian.com/software/jira>

²<https://www.datanyze.com/market-share/project-management--217/jira-market-share>

³<https://enlyft.com/tech/products/atlassian-jira>

the developers team member of the project.

Jira differs from GitHub in that it has its own priority field. This meant that issues are more likely to contain a priority tag. Atlassian, the creator of Jira, suggested a set of five priority levels as the default with the descriptions in Figure 2.1 For each Jira project, the priority levels can also be customized to meet the needs of the project.

Highest - The problem will block progress.

High - Serious problem that could block progress.

Medium - Has the potential to affect progress.

Low - Minor problem or easily worked around.

Lowest - Trivial problem with little or no impact on progress.

Figure 2.1: Default priority levels and description suggested from Jira Documentation [30]

2.3.2 GitHub issue tracking system

An issues on GitHub contains discussions between developers and other members of the team. GitHub users can post issues with bugs, feature requests, feedback, and more. Some companies use GitHub as their main issue tracking system. In the screenshot Figure 2.2 of a GitHub issue from the rust-lang/rust repository. It has a title, description, comments, and status.



Figure 2.2: Screenshot: Example of a GitHub issues from the rust-lang/rust repository.

It can have none or many labels, so the issues do not need to have a priority label. Flutter uses the labels in Table 2.5 to prioritize the issues. Many repository of well-known of projects such as react from Facebook do not have a priority label on any of the issues in their GitHub repository. Normally, there are few repositories on GitHub that have priority labels. In case where they are used, the different definitions also varies from

repositories to repositories. The most commons seem to be high, medium, and low in different variations. Some also include a priority level that is often called critical or urgent, such as Table 2.5.

Label name	Description	Count
P0	Critical issues such as a build break or regression	670
P1	High-priority issues at the top of the work list	1045
P2	Important issues not at the top of the work list	776
P3	Issues that we currently consider unimportant	1893

Table 2.5: Definition of priority labels used in the flutter/flutter repository and count per labels as of 25.09.2023.

2.3.3 Lack of Labels of in GitHub projects

An illustration is the vscode ⁴ repository, where 562 different people opened 752, as shown in Figure 2.3 new issues in one month. Averaging 25 new issues per day.

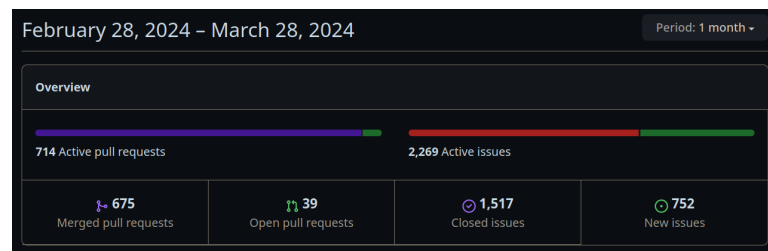


Figure 2.3: New issues created in one month in the vscode repository.

The number of unique people creating new issues per month is 526, shown in Figure 2.4. This suggests that the issues written will likely differ from each other. Even if users follow a template to create issues, the language used can vary. This can make it harder for a transformer model to predict high-priority issues. This highlights the need for models to be trained on a large and diverse set of priority issues so that they can work well on issues created by different users.

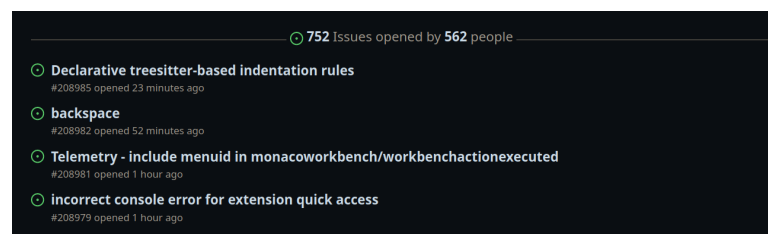


Figure 2.4: Number of unique users creating issues in the vscode repository.

This data was examined on 28 March 2024. At the time of the investigation, there were 7,651 open issues and only 11 of them were marked as

⁴<https://github.com/microsoft/vscode>

important. This suggests that even a large organization cannot keep up with labeling all new issues. For the total number of issues, both open and closed, there were 173,892 issues, with only 2,016 having a label related to priority. This means that only 1.6% of the issues have a priority label.

The priority issue labels were only 2,016, indicating that giving issues a priority level is underused. Note; Microsoft may have some internal tools that I am not aware of. Manually checking all issues takes a lot of time from project owners. In many cases, you would need technical expertise on the subject to determine the priority of the issues. This will reduce the time that a developer can spend coding and implementing solutions.

In this example, an automatic approach to labeling new issues could solve two problems. First, give new issues a priority label. Second, give all the old issues a priority label. If it is a good solution, it depends on how accurate the model is. If the model gives issues the wrong label, it can be confusing and may spend more time correcting the predictions than just manually labeling the issues.

In this example, an automatic approach to labeling new issues could address two key challenges:

1. Assign a priority label to new issues.
2. Assign a priority label to existing issues.

The effectiveness of this solution depends on the accuracy of the model. If the model mislabels issues, it can lead to confusion and potentially require more time to rectify the errors than manually labeling the issues.

Chapter 3

Methodology

This chapter presents an overview of the research methodology, Design Science Research. First, the concept of DSR is introduced, followed by an explanation of the process used to create the final artifact. Then, the creation of different datasets is discussed, including data collection, preprocessing, and data balancing. Finally, the models used are described.

3.1 Design Science Research

Design science research aims to improve human knowledge by creating innovative artifacts and generating design knowledge by solving real-world problems. The goal of a DSR project is to extend human and organizational capabilities by designing novel artifacts, represented as constructs, models, methods, and instantiations. It generates knowledge about how components should be organized or constructed to achieve specific goals, a process known as design knowledge. [4]

3.1.1 DSR process

Brocke et al. [4] suggest a design science research methodology developed by Peffers et al. [21]. fig. 3.1 shows the DSR process model used in this project. The DSR process includes six activities:

1. Problem identification and motivation
2. Define objectives for a solution
3. Design and development
4. Demonstration
5. Evaluation
6. Communication

Each of these activities is explained, along with the steps involved, in the thesis.

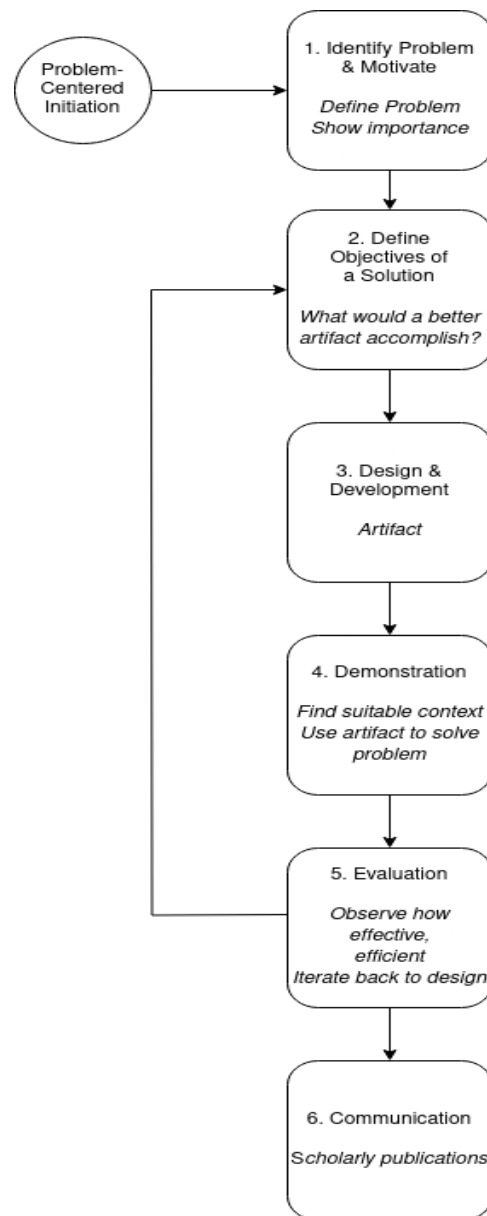


Figure 3.1: DSR Methodology Process model. Adopted from Brocke et al. [4]. Showing each activity in the methodology

There are four possible entry points:

1. Problem-centered initiation
2. Objectives-centered solution
3. Design and development-centered initiation
4. Client/context initiation.

The entry point determines the starting step. This project adopted a problem-centered approach, beginning with activity 1 problem identific-

ation and motivation.

3.1.2 Iterations

In this thesis there were three iterations and a final artifact. The three iterations and the final artifact are shown in Figure 3.2.

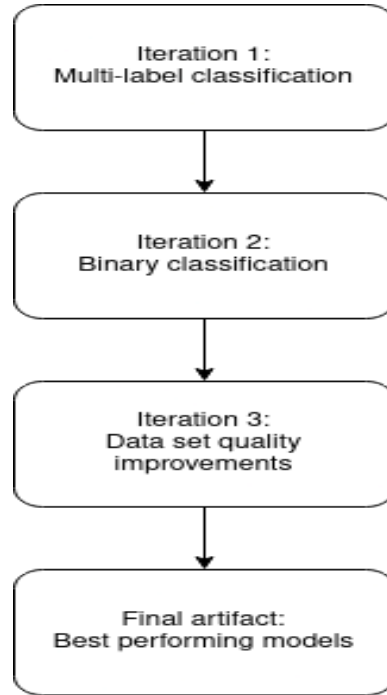


Figure 3.2: The different iterations leading to the final artifact

Multi-label classification: This first iteration explored whether multi-label classification is an effective method of classifying priority of issues.

The second iteration evaluated the efficacy of binary classification and compared it with multi-label classification.

This third iteration investigated the impact of data-enhancement techniques on the performance of the models.

Final artifact. Drawing on the insights from the preceding iterations, the final artifact was developed. It involved testing on projects unseen by the model and fine-tuning with the same projects.

3.2 Activity 1: Problem identification and motivation

This initial activity defines the specific research problem and justifies the value of a solution [4]. This step is crucial for several reasons. It underscores the necessity for a solution and highlights its relevance to both the researchers and the intended audience. This justification not

only motivates stakeholders to engage and support the research, but also enhances the audience's comprehension by clearly articulating the issue at hand. Furthermore, it requires a comprehensive understanding of the current state of the problem and its implications, which is crucial to directing the focus of subsequent research activities.

3.2.1 Identifying the problem for the thesis.

To identify the problem, I participated in discussions with my supervisors and reviewed their previous work. They had developed a model for automatically identifying technical debt in GitHub issues [25], which yielded promising results. The aim was to extend this methodology to other types of labels, such as priority, test, and bugs. In the end, I focused on the priority of issues, recognizing the critical need to find the issues that require the most attention.

After this decision, the first step was to analyze the data set to determine if there was sufficient data to train models on technical debt prioritization. It became clear that there were too few issues labeled both technical debt and priority. It was less than 500 issues, which is too little to train a transformers model. As a consequence, the scope was broadened to include the prioritization of all issue types, given the large number of issues with priority labels. This shift in focus prompted me to investigate various methods of prioritizing issues, which are detailed in Chapter 2.

3.2.2 Gap in in the research

The literature review revealed several unexplored approaches in the field. Two major gaps stood out: first, most existing methods of prioritization focus solely on bugs; second, none has employed transformers to analyze issue descriptions. This is outlined in Section 2.1.5. This discovery prompted me to investigate how transformers could be used to prioritize issues through text classification of the description and title, offering a novel contribution to the field.

3.2.3 Implication of solution

If the results of the solution are positive, several aspects can be significantly affected. The solution can improve efficiency by automating processes, transitioning from manual to automatic labeling of issues. This automation frees up time previously spent on these activities. Additionally, this could lead to cost savings, as fewer resources are required to achieve the same goals, which streamlines operations.

Accurately assigning priority labels to issues requires extensive knowledge about the project, often limiting this task to a select few individuals. Automation minimizes the risk of errors common to manual processes, leading to more reliable results. Automated systems ensure consistent

results, which are essential for processes that need uniformity and adherence to standards.

As the number of issues increases, the time that humans spend also increases. A model can handle a large volume of issues in a much shorter time period. An automated model is not only more scalable but also capable of adapting to various scenarios, making it highly effective across different operational contexts.

3.2.4 Stakeholder of the final artifact

There are several potential stakeholders for the final artifact, including software developers, project managers, and end users of a system.

Software developers are primarily interested in efficient and accurate issue management systems that streamline their workflow, allowing them to focus on coding rather than administrative tasks. They will benefit from automated prioritization, as it reduces the burden of assessing and categorizing new issues. The proposed model would allow developers to see high-priority issues, ensuring that high-priority bugs or features are quickly addressed. This could lead to better management of development cycles and potentially higher quality of the software.

Project managers need to ensure that project timelines are met, resources are optimally allocated, and team productivity is maximized. They are particularly interested in tools that provide a clear overview of project issues and their priorities. Automating the issue prioritization process helps project managers plan and distribute tasks more effectively among team members. It also improves the ability to make decisions about resource allocation based on the importance of pending issues.

QA engineers, who focus on identifying defects in software products before they reach consumers, would also benefit. They are interested in a prioritization system that flags critical issues that could impact product quality. With automated issue prioritization, QA teams can quickly get to work on high-priority problems, improving the overall quality assurance process and reducing the likelihood of critical issues affecting the end-user experience.

Although not directly interacting with issue tracking systems, end users are affected by the efficiency and effectiveness of these systems. They are interested in stable and functional software with minimal bugs. Effective prioritization of issues accelerates the resolution of significant bugs and enhancements, directly improving the user experience. Users benefit from more reliable and better performing software, which in turn can lead to increased user satisfaction and loyalty.

3.2.5 Practical Use Case

The models created could be integrated into issue tracking systems, such as Jira or GitHub. By automating the prioritization process, the model helps

developers and project managers efficiently identify critical issues, thus enhancing the project workflow and optimizing resource allocation.

For example, the model could be utilized in a GitHub Marketplace app or integrated in Jira, where it analyzes each new issue upon creation. This would enable developers to quickly determine the appropriate priority label, thereby streamlining the issue management process. This automation ensures that high-priority issues are flagged as soon as possible, allowing teams to allocate resources more effectively and address critical problems faster. This could potentially be extra beneficial for open source issues with a large number of new issues daily. This can free up developer resources that can be used for other things. This also makes sure that important issues are not overlooked due to a high amount of issues.

In addition, the solution can be particularly valuable in projects with few or no previously prioritized issues. It can predict priority labels for existing issues, potentially uncovering critical issues that had been overlooked. This capability would be especially beneficial in large projects where manual issue assessment is impractical due to the volume of ongoing submissions. By providing a systematic approach to issue prioritization, the model helps to maintain the focus on strategic project goals and ensures that vital tasks are not delayed.

Before involving stakeholders in the evaluation of the artifact, the results must be good enough to not waste their time. Since it is not useful to present results that do not have meaningful impact to the stakeholders. Effort should be rather spent trying to improve the final artifact. Which is the case in the thesis, The results were deemed not good enough to involve stakeholders.

3.3 Activity 2: Define the Objectives of a Solution

Defining the objectives of a solution involves identifying clear and specific goals based on the problem initially identified and understanding the achievable results [4]. The objectives can be categorized as either quantitative or qualitative, each serving distinct purposes in evaluating the success and impact of the proposed solution.

3.3.1 Objectives of the Thesis

For this thesis, the objectives are quantitative. Quantitative goals focus on measurable outcomes that can be directly linked to the performance and efficacy of the model. The main objective of the thesis is to determine whether transformer models can support automatic prioritization of issues in issue trackers such as Jira and GitHub, which is the main RQ as described in Section 1.1.

To assess how well the models perform, several metrics are used. The primary metrics to evaluate the model are MCC, accuracy, precision, recall,

and the F1 score, which are explained in more detail in Section 3.6.1. These metrics provide a comprehensive assessment of the models' performance in prioritizing issues based on their priority.

The metrics can also be used to compare the models' performance with other work that has employed traditional machine learning methods. Using these metrics allows for an objective assessment of the models' performance. However, it is challenging to directly compare performances across different projects as not all use the same datasets and the methodologies may vary slightly. This makes it important to be able to evaluate the metrics based on their score. The different metrics and their implications are explained in detail in Section 3.6.1.

To support the thesis objectives, the performance of the models must be evaluated in real projects to determine if further fine-tuning will enhance performance. To address the main objective, several sub-objectives/sub-research questions were established, which are described in Section 1.1. The results of these sub-RQs will help answer the main objective of the thesis.

RQ1.1: How well do Transformers perform for priority classification when the model is trained and tested on the same dataset? Section 1.1

The objective is to evaluate how well the models perform on a test set from the training dataset. This will provide an indication of the models' effectiveness. The metrics MCC, accuracy, precision, and recall will be used to assess the goals and determine if the performance is good enough to support automatic issues prioritisation. They can also be used for comparative analyses with previous work.

RQ1.2: How well do Transformers generalize to classify priority in projects unseen by the model? Section 1.1

This research question investigates how well the models perform on projects unseen by the model. It is crucial to assess whether the model can generalize to different types of project, since this will give insight how well the models perform in real world scenarios. The same metrics will be used to evaluate whether the models are robust enough and compared to each other.

RQ1.3: How well do Transformers perform for priority classification when the model is fine-tuned? Section 1.1

This research question examines whether fine-tuning improves the models' performance. The same test set will be used for fine-tuning. The process involves using the GTD and incorporating a portion of the test data set to observe the model response and potential performance improvements.

A qualitative goal could be to see if the models save a developer time by having automatic prioritization of the issues. But this depends on the performance of the models, and in this thesis the results were not deemed good enough to investigate this.

3.4 Activity 3: Design and development

In this activity, the artifact is designed and developed. The artifacts are the transformer models. Here, the desired functionality of the artifact is determined. This involves specifying what the artifact should do, its features, and how it should perform to address the identified problem and meet the defined objectives. Creating the artifact where the conceptual design is translated into a product that can be interacted with or evaluated [4]. This phase translates theoretical knowledge into a practical solution.

In this project, the artifact is the machine learning model. Each model can be evaluated by the accuracy, precision, recall and F1 score metrics and the ability of the model to generalize to unseen project. After each model is trained, we get instant feedback and can compare it with the previous models. The final artifact is the best performing model. The way in which the models were designed and developed is explained in Section 3.7.

3.5 Activity 4: Demonstration

The developed artifact is used to demonstrate the solution of one or more instances of the problem [4].

In the thesis, this step was not performed since the results were not good enough to involve and demonstrate the models to potential stakeholders.

3.6 Activity 5: Evaluation

The evaluation activity critically assesses the effectiveness of the artifact created in addressing the defined problem [4]. The observed results from the artifact will be compared to the objectives of a solution.

In this project, we will receive instant feedback on how the model performs after each model is trained, which will be the main way the artifacts is evaluated. The evaluation of the models includes the accuracy, precision, recall, MCC, and F1 score metrics.

3.6.1 Evaluating the Models

This section describes the methods used to assess the performance of models in the final artifact designed to classify high-priority issues in issue tracking systems. The evaluation focuses on determining the accuracy of the models under controlled conditions and their robustness in real-world scenarios.

The performance assessment is conducted through three distinct methods:

- Utilizing the training dataset to evaluate initial model performance as to answer RQ1.1 Section 1.1.
- Testing models in projects they have not previously seen to examine how well they generalize, to answer RQ1.2 Section 1.1.
- Test the adaptability of the models through fine-tuning with data from new projects in the wild, to answer RQ1.3 Section 1.1.

These methods collectively ensure that the models are not only theoretically sound but also effectively applicable in diverse operational contexts.

RQ1.1: N-fold cross-validation and inference in the holdout test set from training data

To address RQ1.1 Section 1.1, a three-fold cross-validation was used along with a holdout test set. Cross-validation is a method to estimate the test error rate [15].

Initially, the entire data set is split into two parts: a training set comprising 85% of the data and a hold-out test set with the remaining 15% of the data. The hold-out test set is kept out during the model training phase. Its primary role is to serve as new, unseen data to evaluate the model after the training and cross-validation process.

The training set is then used for k-fold cross-validation. This involves further splitting the training set into k smaller folds [15], in this case 3. During each of the k iterations, one fold serves as the validation set to assess model performance, while the other k-1 folds act as the training data. Each fold is used once as the validation set, ensuring that the model is trained and validated across all data points in the training set.

A three-fold configuration was chosen to strike a balance between computational demand and the robustness of evaluation metrics. Using more folds, such as 10, would require proportionately more resources and processing time, potentially monopolizing shared computational facilities. This approach significantly reduces the computational burden compared to the higher k-folds, which is crucial given the complexity and size of the data set involved. At the same time, it ensures sufficient data utilization for training the models; each fold uses two-thirds of the data for training, providing a solid basis for assessing model performance while maintaining a manageable variance in the results. This balance is particularly beneficial in scenarios where rapid model iteration is necessary alongside a reliable estimate of performance.

After the model has been trained and validated using k-fold cross-validation, the final model is evaluated using the holdout test set that was set aside in the beginning. This step is crucial because it tests the performance of the model on unseen data, providing an indication of the generalizability of the model.

Although cross-validation helps in understanding how the model performs on different subsets of the training data, a separate test set that the model has never seen during training provides a more unbiased evaluation of its generalization to new data and helps to ensure less bias in the model.

It also helps to prevent overfitting [15] and ensure that the fine-tuning and selection of the model still hold up when applied to completely unseen data. The model can often adapt too closely to the specific nuances of the training data through cross-validation.

A holdout test set also provides a robust check against the variability in model performance that can sometimes result from the random selection of folds in cross-validation. The test set acts as a final check to confirm the findings of cross-validation.

Using both a holdout test set and cross-validation is particularly advantageous in research settings where an unbiased and thorough evaluation of the model is necessary. This approach maximizes the use of available data for training while ensuring rigorous testing and validation, thus balancing the needs for model training and the imperatives of model evaluation.

RQ1.2: Test on unseen projects by the model

To answer Research Question 1.2 (RQ1.2, see Section 1.1), six diverse projects were selected. It is important to note that these are not the same as the holdout test set. These projects were sourced from the GitHub and Jira datasets, with three projects from each. The selected projects include:

- From GitHub: Mypy
- From GitHub: Rust
- From GitHub: Flutter
- From Jira: JIRA
- From Jira: MongoDB
- From Jira: Qt

Note that one of the projects is called JIRA which is project from the Jira data set. This selection allows us to test the models across a wide range of projects. Furthermore, it made it possible for the evaluation of the model trained on GitHub data using test sets from the Jira repositories, and vice versa.

The chosen projects represent a diverse array of software development contexts, including varying programming languages, project scales, and priority types. This diversity is crucial for assessing the robustness of the NLP model beyond the specific conditions of the training data.

The practical utility of the model largely depends on its performance across different environments, beyond those on which it was explicitly

trained. Evaluating the model's performance on the test sets directly addresses this aspect, as it assesses how well the model manages data from diverse sources, which may have different labeling standards or priority definitions.

In addition, the feasibility of deploying such models in standard software development workflows can be assessed. Here the models will be tested on their ability to assist in tracking and prioritizing issues across a range of development projects.

RQ1.3. Fine-tuning on projects unseen by the models

To address the requirements of Section 1.1, a fine-tuning was performed to improve the models' ability to accurately classify high-priority issues in issue tracking systems. This process involves adapting the model to the specific characteristics and language used in previously unseen project data.

30% of data from new projects were incorporated into the existing training data set to fine-tune the transformer models. This proportion, based on the findings of Skryseth et al. [25], strikes an optimal balance between improving the performance of the model and conserving computational resources. Data were selected by stratified sampling to ensure a representative mix of issue types in various projects, thus maintaining a learning base.

The robustness of the fine-tuning process was ensured using a 3-fold cross-validation strategy, as detailed in Section 3.6.1. This method helps validate the consistency and reliability of the fine-tuned model across different subsets of data, providing a solid basis for evaluating improvements in model accuracy. The fine-tuning is expected to enhance the models on performance to the project unseen by the model.

Evaluation metrics

Metrics accuracy, precision, recall, and F1 score were used to evaluate the performance of the models. These metrics are the most commonly used [5]. Matthews correlation coefficients (MCC) were also included since Chicco et al. [7] notes this as a better approach to binary classification problems. The reason for including both accuracy and MCC is that accuracy is the most commonly used metric and makes it easier to compare with past and future work, while it MCC is the better one but not used as much in the existing litterateur.

Accuracy [23] measures the proportion of true results (both true positives and true negatives) in the total data set. It is a metric that gives an overall picture of the model's effectiveness across all classes.

Precision [23] measures the percentage of high-priority issues that have been correctly identified (true positives) in all predictions. This metric is critical in environments where the cost of false positives is high, as it

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (3.1)$$

Figure 3.3: Formula for Accuracy

ensures that the issues identified as high priority are truly critical. If a team has limited resources and needs to ensure that every issue they prioritize as high priority genuinely requires immediate action, precision becomes a key metric.

$$\text{Precision} = \frac{TP}{TP + FP} \quad (3.2)$$

Figure 3.4: Formula for Precision

Recall [23] calculates the proportion of correctly identified high-priority issues (true positives) from all classified high priority issues (true positives + false negatives). High recall is essential in scenarios where missing a high-priority issue could result in significant consequences. In large projects or environments where issues can easily slip through the cracks due to the volume of reports, maintaining high recall ensures that critical issues are not overlooked.

$$\text{Recall (R)} = \frac{TP}{TP + FN} \quad (3.3)$$

Figure 3.5: Formula for Recall

The **F1-score** [23] combines both precision and recall through a harmonic mean, providing a single metric that balances both the precision and the recall of the classifier. It is particularly useful when it is important to maintain a balance between precision and recall, such as in classifying issues where both missing an issue and falsely identifying one are costly. Having a good F1-score is ideal for projects where it is essential to maintain a balance between resource allocation and risk management, such as in software used in moderate-risk scenarios like business operations and customer service platforms.

$$\text{F1 Score} = 2 \times \frac{\text{Precision (P)} \times \text{Recall (R)}}{\text{Precision (P)} + \text{Recall (R)}} \quad (3.4)$$

Figure 3.6: Formula for the F1 Score

An emphasis was placed on **MCC**, which is widely recognized as a more

statistically reliable metric for binary classification [7]. Unlike accuracy, MCC takes into account the balance of the classes and provides a more accurate score that reflects performance across the four categories of the confusion matrix (true positives, false negatives, true negatives, and false positives). This is particularly important in data sets where class imbalance might skew the perceived performance of the model. MCC values range

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP) \cdot (TP + FN) \cdot (TN + FP) \cdot (TN + FN)}} \quad (3.5)$$

Figure 3.7: Formula for the Matthews Correlation Coefficient

from -1 to 1. A value greater than 0 indicates that the classifier is better than a random flip of a fair coin, while 0 means that it is not better. This metric is crucial to ensure that the classifier performs well in all categories of data and provides a balanced view of its effectiveness.

Confusion Matrix

A confusion matrix [7] is a crucial tool in machine learning to evaluate the performance of classification algorithms. It summarizes the prediction results of a classifier by presenting the counts of correct and incorrect predictions for different classes in a matrix format. This matrix is particularly valuable for assessing model accuracy and identifying which classes are prone to prediction errors, thus guiding improvements in the model. In addition, the confusion matrix helps to calculate key performance metrics such as precision, recall, and the F1 score. These metrics are vital, especially in scenarios with imbalanced datasets, as they offer a more detailed understanding of model performance beyond simple accuracy. The confusing matrix is shown in Table 3.1

	Predicted negative	Predicted positive
Actual negative	True Negative	False Positive
Actual positive	False Negative	True Positive

Table 3.1: Quadrants of a confusion Matrix. True positive and true negative are the correct answers and the labels we want [7]

When put into context, the meaning of the quadrants in the confusion matrix when classifying high-priority issues is explained in Table 3.2.

Term	Explanation
True Positive	High-priority issues that are correctly identified as high priority.
False Negative	High-priority issues that are incorrectly identified as not high priority. This is critical to avoid because it represents missed important issues.
False Positive	Not high-priority issues that are incorrectly identified as high priority. This can lead to inefficiency as resources might be allocated to less important issues.
True Negative	Not high-priority issues that are correctly identified as not high priority. While beneficial for overall accuracy, it is less critical in scenarios where the focus is on identifying high-priority issues.

Table 3.2: Explanation of Terms in the Context of Priority Issue Classification

3.6.2 Activity 6: Communication

In this final step, all aspects of the problem and the design artifact are communicated to the relevant stakeholders [4]. This thesis includes the results of the design science research conducted.

3.7 Design and development of the final artifact

This next sections gives an overview of the project workflow as seen in Figure 3.8 and correlate with the design and development activity in the DSR process model. Each training run of the models provides instant feedback on how well the model performs. With this information, it is possible to go back to creating different data sets to train the model on.

Each step of the workflow will be presented according to Figure 3.8. To train the models data sets from two sources, GitHub and Jira, was used. In the next sections, each data set will be presented and how it was processed to make them training data sets.

GitHub and Jira have different ways in how they assign priority levels which led to having different ways of creating the different data sets. This led to separating the development process for each, some things like data cleaning are the data same for both.

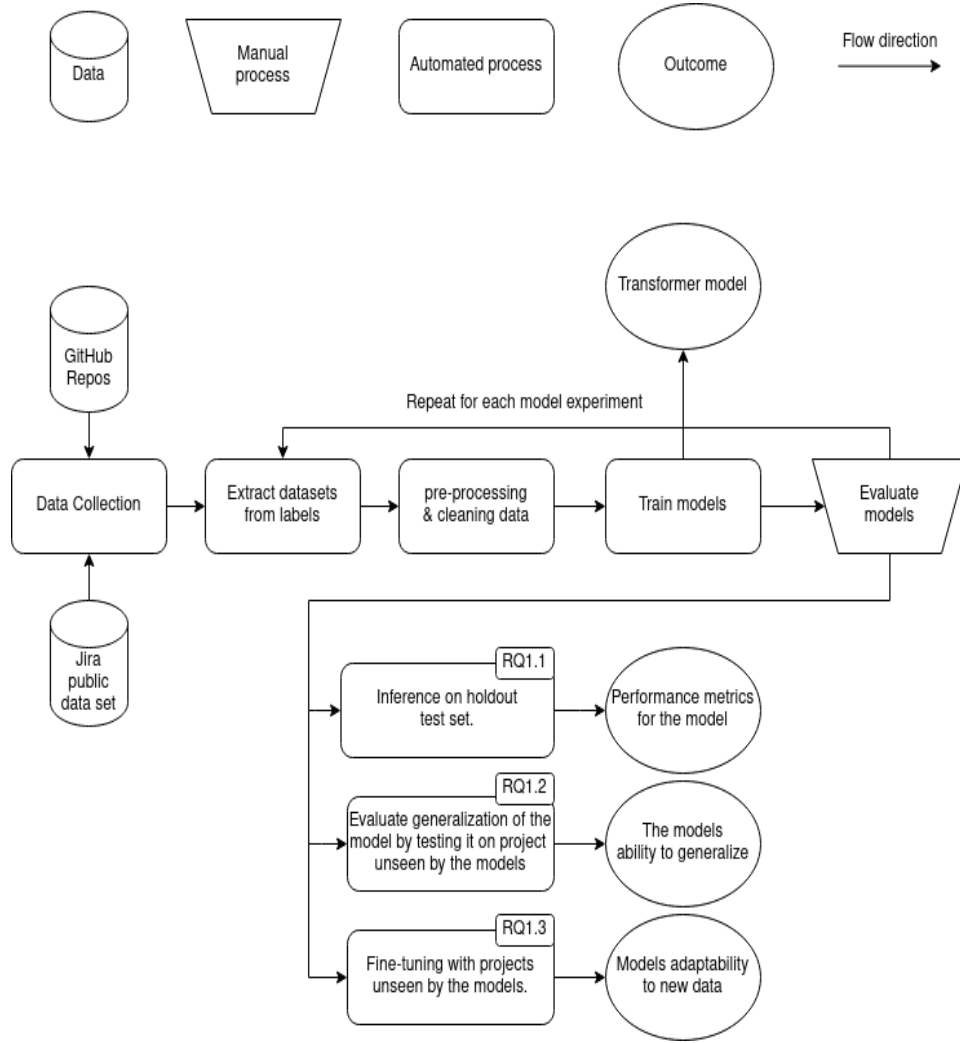


Figure 3.8: This shows the design and development workflow and how the outcomes are related to the research questions.

3.8 GitHub data set

A data set with GitHub issues has been collected and made public by Skryseth et al. [25]. The data set was later updated to include data from 01.01.2015 to 29.09.2023. They mined all data from GHarchives ¹ and extracted all event data to curate this data set. This data include title, description, and labels for each issue. The data set comprises 137 million issues, of which almost 53 million have one or more labels.

3.8.1 Dataset creation and extraction of labels

One of the main challenges was figuring out how to extract all the relevant data to create different training datasets. To create a data set with the

¹<https://www.gharchive.org/>

Total number of	Count
Issues	137,129,493
Issues with labels	52,947,804
Closed issues	47,048,055
Opened issues	67,829,793
Duplicate issues	19,453,335
Unique projects	6,843,296

Table 3.3: Overview number of issues in the GitHub data set.

best possible data, several considerations were made. All labels of a given repository must include all issues with priority levels. This means that if a repository has high, medium, and low priority labels, all issues must be included in the training dataset. This increases the likelihood that issues are correctly labeled. For example, if the repository has high-, medium-, and low-priority labels, the remaining issues do not have priority labels and could be any of the priority labels.

The approach was first to find as many repositories using priority labels as possible. To find repositories, a regex was used to identify repositories using high-priority labels. Then the same issues were extracted from these repositories to form the base of the training data set. After this, all data extracted was from this subdata set. Of the 137 million issues, around 53 million had labels and 7.7 million issues were extracted from 60,238 different repositories with priority labels. The number of priority levels is shown in Table 3.4. 173,979 had a critical priority label and 604,150 a high priority label. This provides plenty of data to work with. Skryseth et al. [25] found around 55,000 technical debt issues and used them to train transformer models with promising results.

After looking at the data there, the most common priority levels were high, medium, and low. This is the priority level many repositories are using just with different names, like mpy. The other most common approach was to include a fourth priority level, called critical, as in the rust repository. One identified problem was that it could be unclear what the different priority means. Since sometimes high is the highest priority levels and other time things like critical is the highest. It is common to set priority levels just using P0, P1 and P2, etc, but the highest priority level could both be P0 and P1, which means that P1 can be both high and critical priority level. From this information, it was decided that the critical and high priority levels were to be combined into just one priority level, called high for the GitHub data set. The large volume of data allowed for some experimentation with removing some issues, such as issues created by bots, which are discussed more in 3.8.2

3.8.2 Preprocessing

The data set was put into four class, high, medium, low and non-priority issues. The data were cleaned as described in Section 3.10. Then, after

Priority Level	Description	Count
Critical	Urgent, critical, blocker, p0	173,979
High	High priority, p1, pri:high	604,150
Medium	Medium priority, normal, pri.med	260,807
Low	Low, trivial, minor	322,452
In Triage	Issues that have not been given a priority	322,172
Non-priority	The rest. Issues with no priority labels	5,907,837
Total number of issues		7,704,706
Number of repositories		60,238

Table 3.4: The number of issues in each priority level and descriptions

the data was cleaned and duplicated and the issues were discarded, the following number of issues was obtained, as seen in Table 3.5

Repositories consisting of many or only bot issues such as the Adguard Filters² have been removed. The reasoning for this is that in this thesis we want to look at issues made by and labeled by humans.

Priority Level	Count
High	552,077
Medium	195,347
Low	219,141
Non-Priority	4,652,524

Table 3.5: Number of issues per class after cleaning in GitHub data set.

From this clean data, two different training datasets were made to be used in the final artifact:

- *all issues vs. high priority*
- *medium and low vs. high priority*

The data set used in the final artifact only included data from the top 50 high priority labels. That means that all repositories using the 50 most common priority levels in GitHub were included in the data set.

3.9 Jira public dataset

The public Jira dataset is a collection of data from 16 public Jira repositories, containing 1,822 projects and 2.7 million issues, as compiled by Montgomery et al. [20]. The data set explained here is used to train the models in final artifact.

²<https://github.com/AdguardTeam/AdguardFilters>

Use of priority levels

Even though most Jira issues have a set of priority levels, the name of the priority level differs a bit from repository to repository. In Jira, a repository is a collection of many projects within an organization. For example, Apache is one repository with 657 projects inside it. The priority levels can be different from repository to repository and also from project to project within the repositories. The number of priority levels varies from four to six levels in different projects.

For example, Spring uses the priority levels: Blocker, Critical, Major, Minor, Trivial for all its projects. Apache has a different priority level for different projects, like: Blocker, Critical, Major, Minor and Trivial, but some projects use: P0, P1, P2, P3, P4 and P5. This led to having to map the priority levels to a common set of labels.

Although most Jira issues are assigned priority levels, these labels can vary significantly between repositories. In Jira, a repository is a collection of projects within an organization; for example, Apache hosts 657 projects. The number of priority levels ranges from four to six in these projects. For instance, all projects within Spring use the levels: Blocker, Critical, Major, Minor, Trivial. In contrast, Apache's projects employ a mixture of these and additional levels such as P0 through P5, necessitating a mapping to a uniform set of priority levels.

Jira has a default set of priority levels [30] as shown in Figure 2.1. So, Highest, High, Medium, Low and Lowest so chosen as priority since it is the default priority levels and it's a bit more intuitive than Blocker, Critical or Major. E.g. Major may sound like one of the highest types of priority but it is the priority level in the middle. Table 3.6 shows an example of how the mapping was done. It also included P1 for blocker and so on. This made the order of the priority levels clear.

MongoDB Priority Levels	Common Priority Level
Blocker - P1	Highest
Critical - P2	High
Major - P3	Medium
Minor - P4	Low
Trivial - P5	Lowest

Table 3.6: MongoDB priority levels mapped to a common set of priority levels.

3.9.1 Data set creation

The steps taken are shown in Figure 3.9. The Jira data set was stored in MongoDB, so the first step was to extract the data from the database and combine it into one dataset. Then, all the issues were mapped to a common set of priority levels, as discussed in Section 3.9. And the test sets Qt,

MongoDB and JIRA were extracted. The data were cleaned as discussed in Section 3.10. This led to a training data set that could be used to create different combinations of classes.

The process is illustrated in Figure 3.9. The Jira data set was stored in MongoDB; the initial step involved extracting the data and consolidating them into a single data set. All issues were mapped to a common set of priority levels as described in Section 3.9. The test sets for Qt, MongoDB, and JIRA were then extracted. The data was then cleaned as detailed in Section 3.10, resulting in a training data set suitable for generating various combinations of classes.

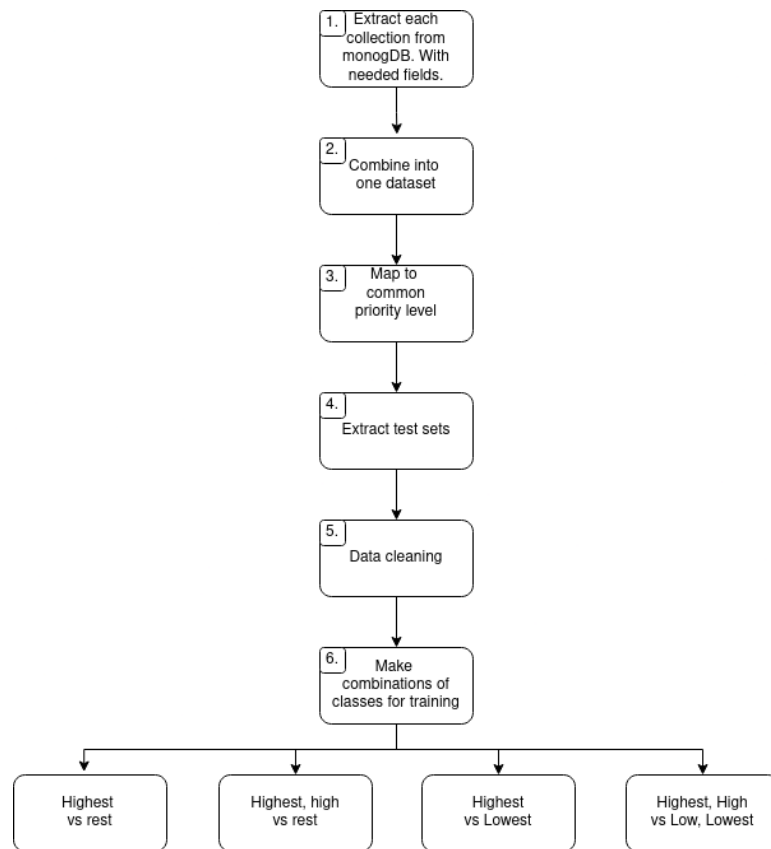


Figure 3.9: Jira data set creation workflow each results in data set used in the final artifact

3.9.2 Jira dataset after cleaning

After the cleaning and removal of the test data, the data set was used to create the training data sets consisting of the following classes as shown in Table 3.7. The bulk of the issues are medium with 933,029. But there are a decent amount of other classes. The highest have 55,358 and the highest 81,404 which are the classes that are the most interesting since the goal is to find high-priority issues. It is also a good amount Low with 267,697 and

Lowest with 32,493. This is also important because there are probably the issues in which it is easiest for the model to find the difference.

Priority Class	Number of Issues
Highest	55,358
High	81,404
Medium	933,029
Low	267,697
Lowest	32,493

Table 3.7: Number of issues per class in Jira data set used to create training data sets after cleaning.

The reasons why most issues are classified as medium can only be speculated on. In contrast, on GitHub, high-priority issues are more common. One possible explanation is that assigning priority levels is a more prevalent practice in Jira. This might result in a conservative approach to prioritization when the appropriate level is unclear. It is possible that issues labeled medium priority could align more closely with high or low priorities, adding confusion to the model. This may explain why medium is the predominant category in the data set.

3.9.3 Including the Jira data Set

After initial experiments with the GitHub data set, the investigation was moved to the Jira data set. An exploration of Jira’s use of priority levels Section 3.9 revealed that Jira has a dedicated priority field, which is frequently used. That means that most of the Jira issues had a priority label set.

This served multiple purposes: It allowed for comparison of the Jira model with GitHub models and provided additional data sets to explore how the GitHub models would perform on Jira data, both in inference and fine-tuning. Binary classification was already determined to be a more effective approach in Section 4.3.4 than multilabel classification; therefore, only the binary classification data set was developed using Jira data.

Table 3.8 show the different data sets used to train the Jira model. Each data set is discussed more in the sections below.

Dataset	Tokens (mean)	Tokens (median)	HP issues	Non-HP issues	Total issues
Jira: Rest vs Highest	84	51	55,358	55,358	110,716
Jira: Rest vs Highest min len 25th	102	64	44,778	44,778	88,556
Jira: Rest vs Highest, High	84	51	136,762	136,762	273,524
Jira: Low, Lowest vs. Highest, High	83	52	136,762	136,762	273,524
Jira: Lowest vs. Highest	75	43	32,493	32,493	64,986

Table 3.8: Summary of dataset token metrics and issue counts in Jira data set.

3.9.4 High, medium, low, lowest vs highest priority

To determine the optimal division of the training class for the model, various configurations were tested and the data set was made accordingly. All data sets are derived from the data set discussed in Section 3.9.2. In the following sections, the different data sets created from the Jira data set are discussed.

The initial configuration examined was rest vs. highest. In this context, the rest of the class has high, medium, low, and lowest priorities. The objective is to identify issues with the highest priority. By categorizing highest into one class and the remaining priorities into another, the model is designed to only detect the highest-priority issues, while the others are considered non-highest priority. In the non-highest priority category, the model is unable to differentiate between high, medium, low, and lowest priorities. This trade-off is made to potentially enhance the model's performance. Although this approach may lead to a more effective model, it comes at the cost of losing the ability to distinguish between high, medium, low, and lowest priorities. However, using multi-label classification gives too poor results as explained in Section 4.3.4.

In Table 3.9, the distribution of issues per class is displayed, showing that the highest priority class contains 55,358 issues, which matches the combined total of the other categories, creating a balanced data set.

Priority Class	Number of Issues
Highest	55,358
Medium	39,259
Low	11,262
High	3,411
Lowest	1,426

Table 3.9: Jira: high, medium, low, lowest vs highest class distribution.

The rest class was undersampled to equal the number of highest priority issues, with the proportions within this class adjusted to reflect their distribution in the entire dataset. This maintains realistic training conditions by mirroring practical scenarios where priority classes are not evenly distributed. By training the model on a dataset that closely resembles the typical priority distribution, the insights and patterns it learns are likely to be more relevant for practical decision-making.

Maintaining this proportional representation helps to evaluate the model's performance more accurately during testing and validation, reflecting how well the model can perform across the naturally distributed distributions it will encounter in a practical scenario. This approach prevents the model from overfitting to particular priority levels that may be overrepresented in a uniformly distributed dataset, thus maintaining balanced sensitivity across various classes.

The data set split focuses on prioritizing the identification of the highest-priority issues, which enables quicker resolution of the most critical problems, reducing the overall impact on system operations or user satisfaction. This binary classification simplifies the model training process, leading to faster convergence and easier optimization compared to a multi-class setup. With a balanced dataset, performance metrics such as accuracy and F1 score become more meaningful and reliable, allowing for a fair evaluation of the model's effectiveness in distinguishing between the most and least critical issues.

3.9.5 High, medium, low, lowest vs Highest min length 25th percentile

This set was done to see if removing some of the shorter issues has an impact on the performance of the model. The same data set used in the Rest vs. Highest Section 3.9.4 analysis was used here to examine the effects of excluding issues shorter than the 25th percentile on data performance. The 25th percentile was chosen after experimentation with the GitHub dataset to strike a balance. The goal is to remove enough issues to perform while retaining sufficient context to generalize to other datasets. This percentile was selected because it eliminates the shortest, often less informative issues without discarding too much data, thereby preserving the robustness of the models' training process.

The objective was to determine whether shorter issues could influence model performance. Shorter issues may present contextualization challenges for transformer models because these models rely heavily on contextual clues to generate predictions. In the case of shorter texts, the lack of extensive contextual data can lead to poorer performance, since the model has fewer tokens from which to learn relationships. This exclusion procedure was performed after data cleaning to ensure that the data used to determine cut-offs were free of errors or outliers that could skew the percentile calculation. And since the cleaning process removes characters that don't provide context for the model, like tags. Performing it before class balancing was crucial to accurately assess the distribution of issue lengths and to ensure that the balancing process would not be influenced by an overrepresentation of shorter, less informative issues. Figure 3.10 displays the issue distribution up to a length of 5000 characters.

The cutoff point of the 25th percentile was 149 characters. Before, the total number of issues was 1,369,981 and after 1,028,415 leading to a removal of 341,566 issues.

3.9.6 Medium, low, lowest vs highest, high

This was done to see how grouping the highest and highest priority levels would impact the results. High was shifted to the highest class and then the two classes were Medium, Low and Lowest vs. Highest and High. This experiment had the same objectives as the rest vs. the highest Section 3.9.4,

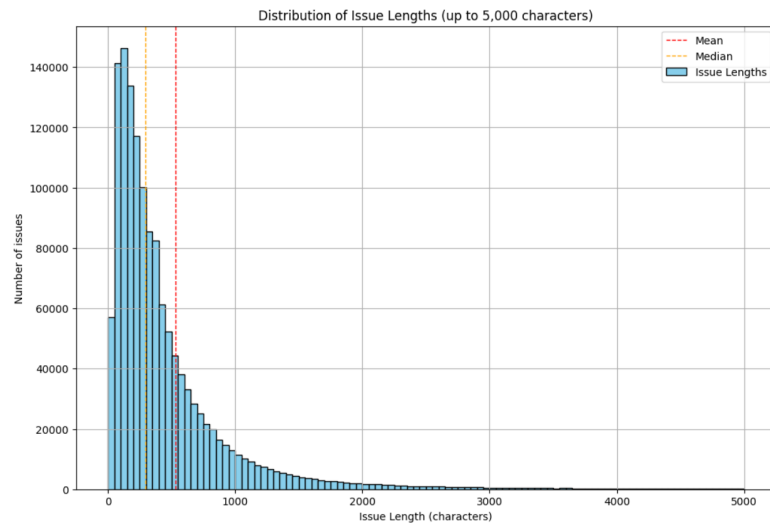


Figure 3.10: Jira: high, medium, low, lowest vs. highest priority with minimum length of the 25th percentile: Distribution of issues length

Statistic	Before	After
Number of Issues	1,369,981	1,028,415
Mean (characters)	530.42	677.11
Standard Deviation	3,850.46	4,434.36
Minimum Length	1	149
25th Percentile	149	251
Median (50th Percentile)	294	394
75th Percentile	557	685
Maximum Length	2,281,081	2,281,081

Table 3.10: Issue lengths summary Statistics before and after removing the lowest 25th percentile.

Class	Count
Highest	44,778
High	31,133
Medium	31,159
Low	9,487
Lowest	999

Table 3.11: Jira: high, medium, low, lowest vs highest 25th minimum length class distribution

but included more priority levels for the highest priority class. This also allowed for more issues in the highest priority class. Speculating that this could improve the performance of the model.

The highest priority level has 55,358 issues and high 81,404. So, this more

than doubles the number of issues in the highest class. The priority levels from the rest class were also increased to match the total number in the highest class with 103,403 medium, 29,743 low and 3,616 issues from the lowest priority class.

Class	Count
Highest	55,358
High	81,404
Medium	103,403
Low	29,743
Lowest	3,616

Table 3.12: Jira: medium, low, lowest vs highest, high class Distribution

Having more issues in the highest priority class allows the model to capture more high priority issues. But it also faces some other problems, since now the model cannot find the difference between highest and high priority issues. This means that the model will potential find more high-priority issues, but it cannot tell which issues have the highest priority. So, ideally, the rest vs. highest model would be preferred if the results are good enough, since here the model finds issues with the highest priority which are the most important ones to identify.

3.9.7 Lowest vs Highest priority

This test how the model would be affected when only presented with the highest and lowest priority issues. This was based on the speculation that it is hard to give mediums issues a correct priority level. This is more a theoretical test to see if the model can correctly differentiate between different priority levels. The highest and lowest were chosen because they should be the easiest to identify. If the results here are promising, at least it shows that the model can find the different between priority levels but struggle when the priority levels are close to each other.

Class	Count
Highest	32,493
Lowest	32,493

Table 3.13: Jira: lowest vs. highest priority class Distribution

3.9.8 Low, lowest vs highest, high priority

Here Low and Lowest vs. Highest and High are tested leaving the medium issues out of the training. This is again to see how different priority levels would affect the performance of the model. Here, only the medium priority level is left out and it is speculated that this could be the hardest priority

level for the model to correctly identify. In many cases, medium-priority issues could lean both as high-priority and low-priority issues.

By comparing the extremes of issue priority, the model can better learn to recognize the features that differentiate the highest from non-highest issues.

Medium-priority issues often represent a gray area in terms of urgency and importance. They can have characteristics of high- and low-priority issues, which can confuse the model during training. By excluding these ambiguous cases, the training process becomes more straightforward, allowing the model to learn more definitive patterns of what clearly defines the highest and lowest ends of the priority spectrum.

But this is not practical since then the model does not know what to do when the encounter mediums issues in a real-world project. This is more of a test to see if the model can differentiate between high- and low-priority levels.

Class	Count
Highest	55,358
High	81,404
Low	122,041
Lowest	14,721

Table 3.14: Jira low, lowest vs highest, high priority: Class Distribution

3.10 Preprocessing and data cleaning of both Jira and GitHub data sets

In this section, the preprocessing and data cleaning processes for both Jira and GitHub are explained, as these processes are nearly identical. Figure 3.11 illustrates the data cleaning process used to prepare and create the data set used to train the models, each step is explained in detail according to the figure.

Removal of test project and bot repositories

The first step was to remove the test set projects, which is later used to run the inference. The test projects are removed from the training data set to ensure that the model is evaluated with previously unseen data. This will assess the generalizability of the models. It will also prevent overfitting, since it will highlight if the models perform on new unseen projects. Inference on unseen project may not perform as well as inference on the hold-out test set. This approach better tests the accuracy of the models in real-world scenarios.

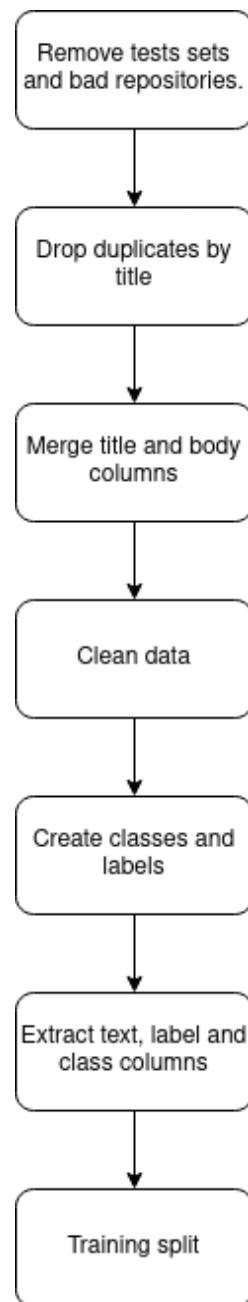


Figure 3.11: Preprocessing and data cleaning process.

In the GitHub data set, projects with only or predominantly bot-created issues were removed. The main reason for this is that we are interested in assessing the human-created issues, not those generated by bots.

This was a manual process in which the top 50 projects with the highest number of high-priority issues were examined. This process was repeated until none of the remaining top 50 projects contained bot-created issues. It was relatively straightforward to identify issues created by a bot, as the creator of the issues hasbot in the name, as shown in Figure 3.12.

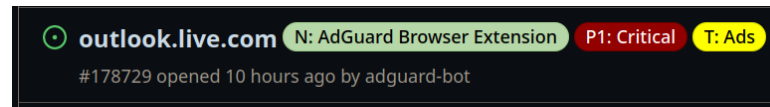


Figure 3.12: Example of bot created issue in AdguardFilters GitHub repository⁴

From the screenshot fig. 3.13 we can see that the issue is created by a bot and the labels given are done by a bot. The issue also includes a screenshot of what the issue is and some comments about it and other information about the system in which the user encountered.

After looking at many other issues, they all follow the same pattern. This can lead to some concerns if they are included in the training data set. Like overfitting to these templates and the issues not inducing enough nuance for the model to learn from.

Data preprocessing

All duplicate issues were removed to ensure data quality and prevent overrepresentation of any particular characteristic, which could lead to model bias. Removing duplicates also improves the efficiency of the training process by reducing unnecessary computational load. Issues without descriptions were also excluded from the dataset, as they lack sufficient information for meaningful analysis and model training.

The titles and descriptions of the issues were merged to create a richer context for each entry. This combination is crucial because titles often contain concise summaries of the issues, while descriptions provide detailed information and context. By merging these two text fields, the model can better understand each issue, obtaining more accurate and context-aware interpretations. This input is particularly beneficial for training transformers, which are capable of understanding complex patterns and dependencies in text data.

Data cleaning

The text was then cleaned using a custom function designed to remove irrelevant information that could distract the performance of the model. The cleaning function reduced all text to lowercase, removed whitespace, and removed special characters and emojis that do not contribute to the meaning of the texts. URLs, HTML tags, and numerical strings intermingled with letters were also removed to focus the models' attention on textual content. The purpose of this cleaning was to distill the text to its most informative elements, enhancing the models' ability to learn from linguistic patterns rather than noise such as HTML tags.

⁴<https://github.com/AdguardTeam/AdguardFilters/issues>

⁶<https://github.com/AdguardTeam/AdguardFilters/issues>

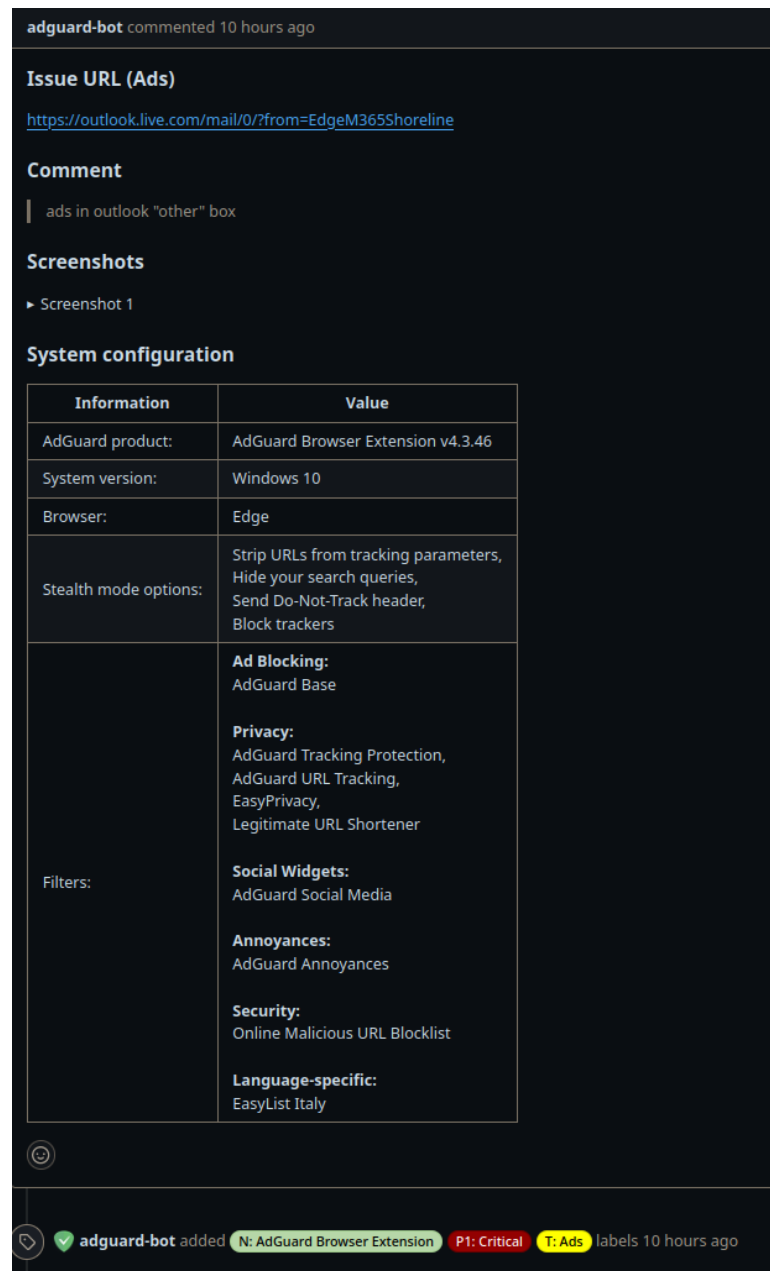


Figure 3.13: Example of a description of an issue created by a bot in AdguardFilters GitHub repository⁶

Label Encoding

Label encoding [18] was used to convert categorical labels into numerical values. This technique is useful for transformers since it requires a numerical input.

For binary classification, this is particularly useful since here you deal with 0 and 1. This simplifies the implementation, and giving 1 will indicate the class we want to find.

The labels used are 1 for the highest priority issues and 0 for others. 1 was consistently assigned to the classes with the highest priorities. An example is that 1 includes the classes highest and high priorities. Then 0 was assigned to the remaining classes: medium, low, and lowest. This classification strategy is explored in greater detail for each created training dataset. Using 1 and 0 simplifies calculations during model training, particularly in functions that measure error and adjust weights, which are critical for optimizing model performance. This binary encoding allows the model to easily differentiate between the two classes, ensuring a clear and efficient learning pathway.

Then only the relevant columns were extracted to form the bases for the basis of the training sets, from this data set the different combination for training can be made. After they are split into different classes, they need to be balanced. The number of issues in each class is not even.

Data set balance

To balance the data sets, undersampling in the majority class was used in all data sets. A random amount of the majority class was sampled to match the number of issues in the minority class. The minor class was in many cases the high-priority class. So, just using a random amount from the non-high class seemed sufficient.

This technique addresses class imbalance, which can improve model performance and prevent the majority class from dominating during training. It also simplifies the training process by reducing the size of the data set, decreasing computational costs, and training time. Furthermore, undersampling can create more robust models by compelling the classifier to focus on the more challenging, often rarer cases, thereby potentially increasing the model's sensitivity to minority class examples. In addition, it can mitigate overfitting by using a more balanced training environment where the learned patterns are representative of all classes, not just the prevalent one.

Tokenization

Tokenization is a process that involves breaking down text data into smaller units known as tokens, which can represent words, phrases, or characters [27]. This step is essential for converting raw text into a format that machine learning algorithms can effectively understand and process. Each token generated through tokenization serves as a feature that machine learning models can utilize to identify patterns and make predictions accurately.

The AutoTokenizer from the Hugging Face transformer library was used for tokenization [28]. This tokenizer is specifically designed to convert text into a format that the model can understand, including splitting the text into tokens and converting these tokens into numerical IDs that the model can process.

The AutoTokenizer automatically loads the tokenizer saved with the DeBERTa-v3 model, ensuring that the tokenization scheme matches the one used during the model’s pre-training. This alignment is crucial for leveraging the model’s learned embedding effectively. The tokenizer also pads or truncates the text to a fixed length required by the model for batch processing[28].

In practice, this step transforms the raw cleaned text from the dataset into a sequence of token IDs, with additional tokens added as needed to represent special elements such as the start and end of sentences. This process is vital for model training, as it directly affects the model’s ability to learn from the textual data by capturing contextual relationships within the input sequences.

3.11 Models used and model selection

To train the model two different models were used. To do the experiment to find the best approach, find the best use of the dataset, DistilBERT was used. The final training of the model and the fine-tuning of the DeBERTaV3 models were used. This was used both to see if a better model was used and to obtain the best possible results.

3.11.1 DistilBERT

DistilBERT, a distilled version of the BERT model introduced by [24], is developed by Hugging Face. It refines BERT’s capabilities through knowledge distillation, reducing the model size by 40% while retaining 97% of BERT’s language understanding capabilities and improving inference speed by 60%. DistilBERT employs a triple loss that includes language modeling, distillation, and cosine distance losses during its pre-training phase.

This optimized model requires less computational power and excels in various NLP tasks. Using knowledge distillation in both pretraining and fine-tuning phases, DistilBERT maintains the flexibility of larger models and performs efficiently under constrained computational budgets. This makes DistilBERT an ideal candidate for thesis experimentation, saving time, computational resources, and energy, thus facilitating more extensive experimentation.

3.11.2 DeBERTaV3

DeBERTaV3 [11] enhances the DeBERTa model by adopting ELECTRA-style training, specifically using replacement token detection instead of masked language modeling. This approach substantially improves sample efficiency during pretraining. DeBERTaV3 also introduces Gradient-Disentangled Embedding Sharing, which optimizes the sharing of token embeddings between the discriminator and the generator, mitigating the

conflicting gradient issues seen in traditional embedding sharing. This method has demonstrated superior performance on various NLU benchmarks, achieving state-of-the-art results, especially in multilingual settings with the DeBERTaV3 model. The detailed architectural adjustments and innovative training strategy have allowed DeBERTaV3 to outperform its predecessors and other contemporary models in efficiency and effectiveness in various NLP tasks.

Skryseth et al. [25] tested three different pre-trained language models, BERT, RoBERTa, and DeBERTaV3. They found that DeBERTaV3 outperforms the other and is also a good indicator for using this PLM. This underlines the finding by He et al. [11].

Chapter 4

Iterations

This chapter describes the iterations that led to the final artifact. It will detail the objectives, implementations, results, and learnings of each iteration. Three iterations were performed, and the findings from these iterations were used to form the final artifact. The three iterations and the final artifact are described as follows:

1. Multi-label classification: This iteration explored whether multilabel classification is an effective method of classifying priority issues.
2. Binary classification: This iteration evaluated the efficacy of binary classification and compared it with multi-label classification.
3. Data quality improvement: This iteration investigated the impact of data enhancement techniques on the results.
4. Final artifact. Drawing on the insights from the preceding iterations, the final artifact was developed. It involved testing on projects unseen by the model and fine-tuning with the same projects.

In these iterations, the focus is on answering RQ1.1 Section 1.1 to find the models that will be further tested on projects in the wild and fine-tuning.

4.1 Design and development for the iterations.

These common things for each iteration, the different way the training set was created, will be discussed for each iteration. All iterations used data from the GitHub dataset. The data were extracted using a regex, which classified them into four categories: high, medium, and low priority, as well as issues without a priority label. The data were uniformly preprocessed and cleaned, following an 80/10/10 split for training, validation, and testing. The model used was DistilBERT, which is a fast and cheaper model as discussed in Section 3.11.

To find the performance of each model, a hold out test set was extracted for

each model. This gave immediate feedback to see how well the models performed, and it was possible to compare the results with each other. Accuracy, precision, recall, and f1 score metrics will be used to evaluate the models.

4.2 Iteration 1. Multi-label classification. High vs. medium vs. low priority

This first iteration investigates how good multilabel classification is for classifying issues into high, medium and low labels.

4.2.1 Objectives

The main objective of this iteration is to see if multilabel classifications can be used to prioritize issues into different categories. The results of the model will determine whether the approach can be used in a practical scenario.

4.2.2 Design and development

In this phase, the GitHub data set was divided into three classes: high-, medium-, and low-priority issues. The class distribution is presented in Table 4.1 with 181,924 issues in each class. The data set was pre-processed and cleaned to ensure the consistency and quality of the data, as described in Section 3.10. The data were then split into training, validation, and testing sets using an 80/10/10 split. The DistilBERT model was employed for this iteration, and the hyperparameters were tuned using the validation set. The model was trained in the training set and its performance was evaluated in the testing set using accuracy, precision, recall, and F1 score metrics.

Priority Levels	Number of issues
High	181,924
Medium	181,924
Low	181,924

Table 4.1: Number of issues per class in high vs. medium vs. low.

4.2.3 Evaluation

The model accuracy is 0.59, as shown in Table 4.2. This performance is slightly better than random guessing, which is ineffective. In the high-priority class, both precision and recall are notably low, at 0.59 and 0.61 respectively. The medium class achieves the best precision at 0.62, but the worst recall at 0.51. The low priority class has the poorest precision at 0.57 and the highest recall at 0.63.

Project	Precision	Recall	F1-score	Accuracy
Multi label: High vs. medium vs. low				
High	0.59	0.61	0.60	0.59
Medium	0.62	0.51	0.56	
Low	0.57	0.63	0.60	

Table 4.2: Results of Multi label classification on hold out test set.

The results in Table 4.2 are poor, and none of the classes achieving convincing performance metrics. In a practical setting, such as a real-world software development project, the model performance would likely lead to incorrect prioritization of issues, which could have significant consequences. For example, high-priority issues might be misclassified as low-priority, leading to delays or neglect of critical tasks.

Given these limitations, it is reasonable to conclude that the current model is not suitable for practical use in an industry setting. Therefore, this led to the exploration of alternative approaches to improve the performance of models and make them more reliable for real-world projects. In the next iteration, the investigation of binary classification of priority issues began. This will answer both how good binary classification is and the results can be compared to the results from the multilabel classification.

4.3 Iteration 2: Binary classifications.

This iteration examines whether binary classifications can be used to prioritize issues. It will explore how the different priority levels are divided into classes. The results will be evaluated and compared with the multilabel classification.

4.3.1 Objectives

The primary objective of this iteration is to investigate the effectiveness of binary classification in prioritizing issues. This involves determining the optimal way to divide the classes, such as high vs. medium/low or high vs. the rest. The goal is to see what priority levels the model has an easier time to identify by doing different combinations of binary classifications.

In doing so, the aim is to understand how well binary classification performs compared to multilabel classification. This iteration also seeks to identify the pros and cons of using binary classification versus multilabel classification, providing an evaluation of both approaches.

4.3.2 Design and development

The first step in this iteration was to split the priority levels into different binary classifications for testing. This involved creating several binary classification models, each designed to distinguish between different

combinations of priority levels. This is to see if the models can differentiate between different priority levels. The rest class is an issue that does not have a priority label. The models were divided as follows:

high vs. medium and low priority: This model aims to differentiate between high priority issues and those with medium or low priority. The goal is to identify whether an issue is of critical or less urgent importance.

High vs. rest: In this model, the focus is on distinguishing high-priority issues from all other issues that are not medium or low priority.

Medium vs. high and low: This model separates medium-priority issues from those with high and low priority. The objective is to identify issues that are important but not as urgent as high-priority issues and more significant than low-priority issues.

Medium vs. rest: Here, the model differentiates medium-priority issues from all other issues that are not high- and low-priority.

Low vs. High and Medium: This model aims to distinguish low-priority issues from those with high and medium priority. The goal is to identify issues that are minor or can be easily resolved, as opposed to more significant issues.

Low vs. rest: In this model, the focus is on separating low priority issues from all other issues in addition to high and medium priority.

4.3.3 Evaluation

The results shown in Table 4.3 are somewhat surprising. You would guess that the model would have an easier time with high vs. medium and low than high vs. rest. That is, since when high vs. medium and low are done, all issues have a priority label. But in the rest they have not been given a priority level. Even though rest does not have a priority label assign, they still have a priority, just that it is unknown. So in the rest class, there could be a mix of high, medium, and low priority issues.

The results indicate that the models' performance varies depending on the specific binary classification task. For example, the high vs. rest model achieved an accuracy of 0.70, with a precision of 0.70 and a recall of 0.74 for the high-priority class. In contrast, the high vs. medium and low models had a slightly lower accuracy of 0.68, with similar precision and recall values for the high-priority class.

Interestingly, the medium vs. rest model performed the best overall, with an accuracy of 0.74. This suggests that the model was more effective in distinguishing medium-priority issues from the rest. On the other hand, the medium vs. high/low model had the lowest accuracy at 0.64, indicating that this classification task was more challenging for the model. This makes a bit of sense, since grouping the two priority levels together is probably confusing the model.

Project	Precision	Recall	F1-score	Accuracy
High vs. rest				
High priority	0.70	0.74	0.72	0.70
rest	0.70	0.66	0.68	
High vs. medium, low				
High priority	0.68	0.69	0.68	0.68
medium low priority	0.68	0.67	0.68	
Med vs. rest				
Medium priority	0.76	0.70	0.73	0.74
Rest	0.70	0.77	0.74	
Med vs. high/low				
Medium priority	0.64	0.62	0.63	0.64
high, low priority	0.62	0.66	0.65	
Low vs. rest				
Low priority	0.71	0.74	0.72	0.71
Pest	0.71	0.67	0.69	
Low vs. high/medium				
Low priority	0.66	0.68	0.67	0.66
high, medium priority	0.67	0.65	0.66	

Table 4.3: Result from binary classifications.

For the low-priority models, the low vs. rest model achieved an accuracy of 0.71, with a precision of 0.71 and a recall of 0.74 for the low-priority class. The low vs. high/medium model had a slightly lower accuracy of 0.66, with precision and recall values of 0.66 and 0.68, respectively, for the low priority class.

Overall, the models with the other class performed better than their counterparts. But none of the models performed particularly well and improvements must be made before they are ready to be used in a practical scenario.

The next section will discuss why binary classification was chosen over multi-label classification. Then iteration 3 will be presented with the aim of improving the quality of the data sets.

4.3.4 Why use binary over multi-label text classification

This section outlines the advantages and disadvantages of binary and multi-label text classification with transformers.

Multi-label classification models can handle scenarios that reflect more realistic complexity, where an instance might belong to multiple classes simultaneously. The capability to distinguish between several closely related classes (high, medium, low) can potentially provide richer insights and finer categorical distinctions, which are beneficial for nuanced decision-making processes. However, the performance metrics observed—precision, recall, and F1 scores ranging from 0.51 to 0.63—indicate

poor effectiveness in class differentiation. These scores suggest challenges in the model's ability to accurately distinguish between more than two categories.

Binary classification models demonstrated higher performance metrics consistently around 0.70 across different measures, reflecting a robust ability to distinguish between the high and non-high classes. The binary approach simplifies the training process, as it focuses on differentiating two classes rather than multiple classes, which often results in a more straightforward and computationally efficient model. This simplicity also results in lower data requirements and reduced risk of overfitting, making binary models particularly advantageous in scenarios with limited data resources. However, the primary limitation of binary classification is its reduced capability to provide detailed insights across multiple categories, as it aggregates all non-high outcomes into a single 'non-high' category.

Given the better results, the binary classification approach is chosen because the performance is better. By choosing to use a binary classification approach, sacrifice is made to differentiate between different levels, in favor of better performance. In practice, this means that we find issues with high priority and the rest can be both medium- and low-priority issues.

But finding high priority is the most important thing since these are the issues that need to be taken care of first. So, it should still have relevance to aid in the labeling of issues.

4.4 Iteration 3: Improve the quality of the data set

4.4.1 Objectives

The primary objective of this iteration is to determine whether enhancing the quality of the data set will result in improved model performance. This initiative is driven by the observation that the current model accuracy is insufficient when trained on the full dataset. The goal is to explore a few techniques, see if they have an impact on the results, and to find out which approach is the best.

4.4.2 Design and Development

In this iteration, two different approaches were used to see if they improve the performance of the model. The focus will be on the high vs. rest model, since that is the setup that we are most interested in finding. The first approach was to remove issues with a length under a certain threshold. The second was to only select issues from repositories with at least a certain number of high-priority issues.

Remove Minimum Length on Issues

The goal was to test different lengths by removing issues with fewer than a threshold of characters. Various lengths were tested: 100, 500, and 1000 characters. The objective was to find the optimal length to remove from the dataset while retaining as much data as possible and observing an increase in model performance. The testing focused on high-priority versus random issues, as improving high-priority classification is the primary goal.

Minimum Length of 100 Characters

After cleaning, the number of issues was reduced to 183,705 with a 50/50 balance of classes.

Category	Before	After
Issues removed from high priority	242,229	199,792
Difference	42,437	
Issues removed from rest	1,344,895	1,119,411
Difference	225,484	

Table 4.4: Number of issues before and after removing issues with fewer than 100 characters

Minimum Length of 500 Characters

After cleaning, the number of issues was reduced to a balanced data set. Placeholder numbers are used here and should be replaced with actual data once available. The number of issues after cleaning (with a 50/50 class balance) was 87,396.

Category	Before	After
Issues removed from high priority	242,229	87,396
Difference	154,883	
Issues removed from random	1,344,895	544,158
Difference	800,737	

Table 4.5: Number of issues before and after removing issues with fewer than 500 characters

Minimum Length of 1000 Characters

After cleaning, the number of issues was reduced to 40,221 with a balance of 50/50 classes.

Repository Selection Based on Issue Count

The second approach was to select issues only from repositories that have at least a certain number of high-priority issues. This ensures that the model is trained on repositories with a significant number of high-priority

Category	Before	After
Issues removed from high priority	242,229	41,569
Difference	200,660	
Issues removed from random	1,344,895	297,612
Difference	1,047,283	

Table 4.6: Number of issues before and after removing issues with fewer than 1000 characters

issues, which could provide a more robust training ground for the model to learn and improve its prioritization accuracy. The threshold for the number of high-priority issues required in a repository was set based on initial observations and analysis of the dataset.

In this approach, each repository selected for the dataset contains a minimum of 100 issues labeled high priority. A total of 256 repositories were chosen, excluding three test repositories. To ensure consistency, random issues were selected from the same repositories as high-priority issues, ensuring that they were randomly chosen from all repositories with high-priority issues.

Additionally, the random data set was curated to exclude all medium- and low-priority issues, leaving only high-priority and random issues. The data set was categorized into two classes: high priority and random. After the cleaning process, each class contained 58,401 issues.

By implementing these strategies, this iteration aims to refine the dataset and enhance model performance, contributing to more accurate and reliable issue prioritization in issue tracking systems.

4.4.3 Evaluation

The evaluation of this iteration focuses on the impact of improving the quality of the data set on the performance of models trained with DistilBERT. Approaches such as setting a minimum length for issue descriptions and selecting repositories based on the number of high-priority issues were tested. The results are summarized in Table 4.7.

high vs. Rest, minimum length 100 characters per Issue: The model trained with a minimum length of 100 characters per issue achieved an accuracy of 0.71. The precision and recall for high-priority issues were both 0.70 and 0.71, respectively, resulting in an F1 score of 0.71. Similarly, for the rest of the issues, the precision was 0.71, the recall was 0.70, and the F1 score was 0.70. These results indicate a balanced performance, but suggest that merely filtering out very short issues may not significantly boost model performance.

High vs. rest, minimum length 500 characters per issue: When the minimum length was increased to 500 characters, the accuracy improved to 0.75. For high-priority issues, precision increased to 0.73, recall to 0.77, and F1 score

Project	Precision	Recall	F1-score	Accuracy
High vs. rest, min length 100 character per issue.				
All issues	0.71	0.70	0.70	
High priority	0.70	0.71	0.71	0.71
High vs. rest, min length 500 character per issue.				
All issues	0.76	0.72	0.74	
High	0.73	0.77	0.75	0.75
High vs. rest, min length 1000 character per issue.				
High priority	0.75	0.75	0.75	0.75
All iissues	0.76	0.75	0.76	
High vs. rest, min 100 issues per repositories				
All issues	0.73	0.70	0.71	
High priority	0.70	0.73	0.72	0.72

Table 4.7: Results from models trained with DistilBERT after improving quality of the data.

to 0.75. For the rest of the issues, the precision was 0.76, the recall was 0.72 and the F1 score was 0.74. This suggests that setting a higher minimum length filter helps improve the models ability to distinguish between high priority and other issues.

High vs. Rest, Minimum Length 1000 Characters per Issue: With a minimum length of 1000 characters, the model accuracy remained at 0.75. For high priority issues, both precision and recall were 0.75, resulting in an F1 score of 0.75. For the rest, the precision was 0.76, the recall was 0.75, and the F1 score was 0.76. These results are similar to those for the 500-character filter, indicating that further increasing the length threshold does not significantly impact performance beyond the gains achieved with 500 characters.

High vs. rest, minimum 100 issues per repository: By selecting repositories with at least 100 high-priority issues, the model achieved an accuracy of 0.72. The precision and recall for high-priority issues were 0.70 and 0.73, respectively, resulting in an F1 score of 0.72. For the rest, the precision was 0.73, the recall was 0.70, and the F1 score was 0.71. These results show that focusing on repositories with a substantial number of high-priority issues provides a moderate improvement in model performance.

The evaluation demonstrates that filtering issues based on length and repository selection can enhance model performance. A minimum length of 500 characters per issue offers the best improvement, balancing precision, recall, and overall accuracy. Selecting repositories with a significant number of high-priority issues also contributes positively, though the impact is less pronounced compared to length-based filtering. These findings highlight the importance of data quality in the training of effective models for identifying issues prioritizing.

Chapter 5

Final Artifact

In this chapter, the results and findings of the final artifact will be presented. First, the results of training the models with k-fold cross-validation and the results of inference will be presented to answer RQ1.1, Section 1.1. The models will then be tested on projects unseen by the model to answer RQ1.2 Section 1.1. Finally, the best-performing models will be fine-tuned in unseen projects to answer RQ1.3 Section 1.1.

5.1 Jira: medium, low, lowest vs. highest and High

For this model, lowest, low, medium is in one class, and high and highest are in the other class.

The model was evaluated using a holdout test set and three real-world projects Jira, Qt, and MongoDB. Both the holdout test set and the projects are unseen by the model. The performance metrics of the model are summarized in Table 5.1.

The holdout test set showed an MCC score of 0.406 which is somewhat low. The rest of the metric was all round 0.70 for both classes.

All projects scored less than the holdout test set. Qt had the best MCC score with 0.356, but had a recall for the Highest, High class of only 0.59. The recall for the medium, low and lowest classes was slightly higher, 0.76. However, for precision, it was the other way around with 0.71 for the highest, high, and 0.65 for the medium, low, and lowest class.

MongoDB has an MCC score of 0.327 and all other metrics were close together ranging from 0.65 to 0.67.

JIRA had the worst MCC score with 0.245, but it also had the best recall for the highest and high class with 0.84. For medium, low, lowest, the recall was low with 0.37. For precision, highest and high were 0.57 and Medium, Low, Lowest 0.71.

Dataset	Precision	Recall	F1-score	Accuracy	MCC
Hold out test set					
Medium, Low, Lowest	0.71	0.69	0.70	0.70	0.406
Highest, High	0.70	0.71	0.70		
JIRA					
Medium, Low, Lowest	0.71	0.37	0.49	0.61	0.245
Highest, High	0.57	0.84	0.68		
Qt					
Medium, Low, Lowest	0.65	0.76	0.70	0.68	0.356
Highest, High	0.71	0.59	0.65		
MongoDB					
Medium, Low, Lowest	0.67	0.65	0.66	0.66	0.327
Highest, High	0.66	0.67	0.67		

Table 5.1: Jira: medium, low, lowest vs. highest, high inference on holdout test set

5.2 Jira: lowest vs. highest

To explore the ability of the model to differentiate between priority levels, a binary classification approach was used, focusing on the lowest and highest priority classes in the Jira data set. This experiment aimed to determine whether the model can effectively distinguish between these two extreme classes.

The results of this experiment are presented in Table 5.2. The table displays the performance metrics for the model in the holdout test set, as well as in the Qt, MongoDB, and JIRA projects.

The results show that the model performs well in the holdout test set, with an MCC of 0.679. For the highest class precision, are 0.82, recall 0.87 giving an F1 score of 0.81. The lowest class also scored pretty good with a precision of 0.86, recall of 0.80 and F1 score of 0.83. Showing that the model can find both lowest and highest priority pretty good.

For the projects in the wild, both Qt and MongoDB performed overall pretty well with an MCC of 0.598 and 0.529. Qt had a precision of 0.75, recall of 0.89 and F1 score of 0.81 for the highest class. For the lowest class precision of 0.86, recall 0.70 and the F1 score 0.77. The accuracy is 0.79.

For MongoDB most of the metrics score a bit lower, but they were much closer together. With all the metrics ranging from 0.75 to 0.77.

The test set that performs the worst in the wild is JIRA with an MCC score of 0.362. It performed well on some metrics and poor on others, making it a bit uneven. The recall was notably good for the highest class with 0.90, but the precision was only 0.61 giving an F1 score of 0.72. For the lowest class, it was the opposite, with precision being the best with 0.80 and recall low with 0.42 giving an F1 score of 0.55. The accuracy was 0.66.

Overall, the model performs well on the holdout set and decently on the Qt and MongoDB projects, but unevenly on the JIRA project. This suggests that the model can differentiate between the priority classes that are farthest apart but may struggle when mixing in other priority levels.

Project	Precision	Recall	F1 score	Accuracy	MCC
Holdout test set					
Lowest	0.86	0.80	0.83	0.84	0.679
Highest	0.82	0.87	0.84		
Qt					
Lowest	0.86	0.70	0.77	0.79	0.598
Highest	0.75	0.89	0.81		
MongoDB					
Lowest	0.76	0.77	0.77	0.76	0.529
Highest	0.77	0.75	0.76		
JIRA					
Lowest	0.80	0.42	0.55	0.66	0.362
Highest	0.61	0.90	0.72		

Table 5.2: Jira: lowest vs. highest. Inference on holdout test set and projects unseen by the model.

5.3 Jira: low, lowest vs. highest, high

To further investigate how good the model is for finding different priority levels, another model was trained with only the removal of the medium priority level. Leaving low and lowest in one class and highest and high in the other class. The performance of the model, differentiated by the low, lowest, highest, high priority levels in various projects, is summarized in Table 5.3.

For the holdout test dataset, the model achieved precision scores of 0.79 for low and lowest priorities and 0.76 for highest and high priorities. The recall scores were 0.75 for low and lowest and 0.79 for highest and high priorities, resulting in F1 scores of 0.77 and 0.78 respectively. The overall accuracy across both priority levels was 0.77, with a MCC of 0.544.

The MongoDB project was the best performing test data set. Issues with low and lowest priorities showed precision and recall scores of 0.74 and 0.77, respectively, resulting in an F1 score of 0.75. For issues with highest and high priorities, precision was 0.76 and recall was 0.73, with an F1 score of 0.74. The overall accuracy for this data set was 0.75, with an MCC of 0.498.

In the JIRA project-specific data, the low and low priority issues recorded a precision of 0.74 and a recall of 0.49, leading to an F1 score of 0.59. The highest and highest priority issues had a lower precision of 0.62 but a higher recall of 0.83, and an F1 score of 0.71. The overall accuracy for JIRA was 0.66 with an MCC of 0.341.

For the Qt project, the precision for the low and lowest priorities was 0.72, with a recall of 0.77 and an F1 score of 0.74. The highest and highest priorities achieved a precision of 0.75, recall of 0.70, and an F1 score of 0.72. The aggregate accuracy stood at 0.73, accompanied by an MCC of 0.468.

Project	Precision	Recall	F1-score	Accuracy	MCC
Holdout test data set					
Low, Lowest	0.79	0.75	0.77	0.77	0.544
Highest, High	0.76	0.79	0.78		
MongoDB					
Low, Lowest	0.74	0.77	0.75	0.75	0.498
Highest, High	0.76	0.73	0.74		
Qt					
Low, Lowest	0.72	0.77	0.74	0.73	0.468
Highest, High	0.75	0.70	0.72		
JIRA					
Low, Lowest	0.74	0.49	0.59	0.66	0.341
Highest, High	0.62	0.83	0.71		

Table 5.3: Jira: Low, Lowest vs Highest, High. Inference on test sets

5.4 Summary of results for RQ1.1

This section answer RQ1.1: How well do Transformers perform for priority classification when the model is trained and tested on the same dataset? Section 1.1. This section presents the models trained with DeBERTaV3. This also helps to find how the models react with different types of training dataset configurations.

This section presents the classification performance of various models trained and tested on datasets from GitHub and Jira.

5.4.1 GitHub models

In the GitHub data set *Top50: medium and low vs. high priority*, the model achieved an accuracy of 0.80 and an MCC of 0.600. For medium and low priority issues, precision was 0.77, recall was 0.85, and F1 score was 0.81. High-priority issues showed a Precision of 0.83, Recall of 0.74, and an F1 score of 0.79.

A similar analysis for all issues vs. high priority in the GitHub dataset demonstrated an overall accuracy of 0.76 and an MCC of 0.515. The model classified all issues priority items with a precision of 0.75 and a recall of 0.78, resulting in an F1 score of 0.76, while high priority issues have a Precision of 0.77, Recall of 0.73, and an F1 score of 0.75.

This made the medium and low vs. high priority the best performing GitHub model and it was choose to further testing to answer RQ1.2 inference

on unseen project and R1.3 fine tuning on the unseen projects.

5.4.2 Jira models

The Jira data set was analyzed in various scenarios. In the category Rest vs. highest, both groups had similar F1 scores of 0.72, with an overall accuracy of 0.72 and an MCC of 0.434.

When considering rest vs. highest min length 25th percentile, accuracy slightly decreased to 0.71, and the MCC was 0.412. Here, 'Rest' had a precision of 0.73 and a recall of 0.64, whereas 'Highest' showed a precision of 0.69 and a higher recall of 0.77.

Further assessments include Rest vs. highest, high with an overall accuracy and MCC of 0.70 and 0.406

Lowest vs. Highest, which performed best with an accuracy of 0.84 and an MCC of 0.679. The 'Lowest' priority items achieved a precision of 0.86 and recall of 0.80, while 'Highest' recorded a precision of 0.82 and recall of 0.87.

Lastly, in the low, lowest vs. high, high category, the model maintained an accuracy of 0.77 and an MCC of 0.544. The Low, Lowest group showed a precision of 0.79 and recall of 0.75, and the Highest, High group had slightly lower precision of 0.76 but higher recall of 0.79.

The results here showed that the *high, medium, low and lowest vs. highest* model is the Jira model, which is also a model that can be used in a practical scenario. That is why it was chosen to test the project further and fine-tune the unseen projects. It was also tested to remove all issues under the 25th percentile (200 characters) to see if it had an impact on the results, and it performed worse than keeping all the issues.

Dataset	Precision	Recall	F1-score	Accuracy	MCC
GitHub: Top50. Medium and low vs. high priority					
Medium and Low	0.77	0.85	0.81	0.80	0.600
High	0.83	0.74	0.79		
GitHub: Top50. Rest vs. High priority					
All issues	0.75	0.78	0.76	0.76	0.515
High	0.77	0.73	0.75		
Jira: high, medium, low, lowest vs. highest					
Rest	0.72	0.71	0.72	0.72	0.434
Highest	0.71	0.72	0.72		
Jira: high, medium, low, lowest vs. highest. min len 25th percentile					
High, medium, low, lowest	0.73	0.64	0.68	0.71	0.412
Highest	0.69	0.77	0.72		
Jira: Rest vs. highest, high					
Medium, low, lowest	0.71	0.69	0.70	0.70	0.406
Highest, High	0.70	0.71	0.70		
Jira: Lowest vs. Highest					
Lowest	0.86	0.80	0.83	0.84	0.679
Highest	0.82	0.87	0.84		
Jira: Low, Lowest vs. Highest, High					
Low, Lowest	0.79	0.75	0.77	0.77	0.544
Highest, High	0.76	0.79	0.78		

Table 5.4: Overview of all the different models trained with DeBERTaV3

5.5 Jira: high, medium, low, lowest vs. Highest priority

For Jira this model was selected as the most fitting to do fine-tuning on. It had the best performance of the models that makes practical sense. However, since we are interested in finding issues with the highest priority levels this makes sense.

The data set used to train this model was high, medium, low, and lowest. vs. highest priority. Rest will be also used to denote the high, medium, low, and lowest class from now on. The different priority levels are explained in Section 3.9. This is because we are more interested in finding issues with the highest priority.

5.5.1 RQ1.1: Training and test on holdout test set

The Table 5.6 presents the performance metrics of the DeBERTaV3 model when tested on the hold-out data set, differentiating between the high, medium, low, lowest (called rest from now on) and highest priority classes. The model was tested with the holdout data set during inference, with an 85/15 train / test split and a grouped k-fold cross-validation of $k = 3$. Both classes have similar precision, 0.72 for rest and 0.71 for highest priority, recall 0.71 and 0.72, and F1 score 0.72 for both. MCC for the model is 0.434,

Description	Count
Total issues	110716
Training dataset	94108
Rest issues	47084
Highest issues	47024
Test dataset	16608
Rest issues	8334
Highest issues	8274

Table 5.5: Jira: high, medium, low, lowest vs. highest dataset Distribution

indicating moderate predictive quality. The overall accuracy of the model across both classes is 0.72, suggesting a consistent level of performance in classifying the data set into Rest and Highest priority categories.

DeBERTaV3	Precision	Recall	F1-score	Accuracy	MCC
Rest	0.72	0.71	0.72	0.72	0.434
Highest priority	0.71	0.72	0.72		

Table 5.6: Testing on hold out test During Inference Rest vs Highest priority

5.5.2 RQ1.2 and RQ1.3: Inference and fine-tuning on projects unseen by the model

Inference will answer RQ1.2 Section 1.1. Fine-tuning will answer RQ1.3 Section 1.1. Here, both inference on the test sets and fine-tuning will be discussed.

For inference on the test sets, the result varied. Some projects surprisingly performed better than the holdout test set, while the others performed poorly. This model is trained on the Jira data set. The projects Qt, JIRA and MongoDB are projects from the Jira data set, while Flutter, Rust, and Mypy are from the GitHub. That way we can see how well the model performs on other data sets that they are trained on.

Starting with Jira data sets. The best performing project was Qt with an MCC score of 0.542, which is higher than the holdout test. MCC saw a slight improvement after fine-tuning with 0.570. Most of the other metric also saw a slight improvement with for the rest class precision going from 0.80 to 0.81, recall from 0.72 to 0.76 and F1 score from 0.76 to 0.78. For the highest class, precision went from 0.75 to 0.76, but recall decreased from 0.75 to 0.76. Accuracy also had a slight increase from 0.77 to 0.78.

The rest of the Jira projects, MongoDB and JIRA preformed a bit worse than the holdout test set. Both saw an increase after fine-tuning with the MCC score of MongoDB going from 0.284 to 0.310 and JIRA going from 0.360 to 0.400. The rest of the metric also slightly increased.

For the GitHub projects, there were also some varied results. The best

performing project was Flutter with an MCC of 0.394. This project had the highest increase after fine-tuning with an MCC of 0.605. The accuracy went from 0.69 to 0.80. For the rest class, precision went from 0.75 to 0.82, recall saw a big increase from 0.58 to 0.80 and the F1 score went from 0.65 to 0.80. For the highest class, precision went from 0.66 to 0.79 and recall from 0.80 to 0.83 with an F1 score of 0.72 to 0.80. This was the only project with such an increase after fine-tuning.

For the other GitHub project the inference on the holdout test set performance was poorly, but they saw an increase in performance after fine-tuning. Rust had an MCC of 0.316 that increased to 0.362 and mypy a MCC of 0.282 increasing to 0.333. Most of the other metrics also had a slight increase.

Overall, inference and fine-tuning on the project in the wild shows varied results. If the model is applied to project such as Qt and Flutter then the results show some promising that it can be applied in a practical setting with help assigning the highest priority level to issues. But if used in other projects like MongoDB, it will probably do more harm than good.

Project	Precision	Recall	F1-score	Accuracy	MCC
Holdout test set					
Rest	0.72	0.71	0.72	0.72	0.434
Highest priority	0.71	0.72	0.72		
Jira: Qt					
Rest	0.80	0.72	0.76	0.77	0.542
Highest	0.75	0.82	0.78		
Jira: Qt, 30%					
Rest	0.81	0.76	0.78	0.78	0.570
Highest	0.76	0.81	0.79		
Jira: Jira					
Rest	0.72	0.58	0.64	0.68	0.360
Highest	0.65	0.77	0.70		
Jira: Jira, 30%					
Rest	0.72	0.64	0.68	0.70	0.400
Highest	0.68	0.76	0.72		
Jira: MongoDB					
Rest	0.63	0.70	0.66	0.64	0.284
Highest	0.66	0.58	0.62		
Jira: MongoDB, 30%					
Rest	0.65	0.67	0.66	0.65	0.310
Highest	0.66	0.64	0.65		
GitHub: Flutter					
Medium and low	0.75	0.58	0.65	0.69	0.394
High	0.66	0.80	0.72		
GitHub: Flutter, 30%					
Medium and low	0.82	0.78	0.80	0.80	0.605
Highest	0.79	0.83	0.81		
GitHub: Rust					
Medium and low	0.69	0.58	0.63	0.66	0.316
High	0.63	0.74	0.68		
GitHub: Rust, 30%					
Medium and low	0.73	0.60	0.66	0.68	0.362
High	0.64	0.76	0.69		
GitHub: Mypy					
Medium and low	0.59	0.84	0.69	0.63	0.282
High	0.72	0.42	0.53		
GitHub: Mypy, 30%					
Medium and low	0.60	0.87	0.71	0.65	0.333
High	0.77	0.43	0.55		

Table 5.7: Jira: high, medium, low and lowest vs. highest priority. Inference on projects unseen by the model and fine tuning on them with 30% of data from the project included in the training data set.

5.6 GitHub: medium and low vs. high priority. Top50

5.6.1 RQ1.1: Training and inference in the holdout test set

This model aims to classify GitHub issues into high versus medium and low priority categories. To evaluate its effectiveness, a comprehensive data set comprising 42,630 GitHub issues was used, as shown in Table 5.8. The data set was divided into an 85/15 train/test split with a training set of 36,235 issues and a testing set of 6,395 issues, ensuring that the model was evaluated on previously unseen data.

Description	Count
Total issues	42,630
Training data set	36,235
Rest issues	18,128
Highest issues	18,107
Test data set	6,395
Rest issues	3,208
Highest issues	3,187

Table 5.8: GitHub: med, low vs. high Dataset Distribution

The performance of the model is illustrated in Table 5.9. The model demonstrates moderate performance with an MCC of 0.600 and an accuracy of 0.80. For the high-priority class, the results were: precision 0.83, recall 0.74, and F1 score 0.79. For the medium- and low-priority class, the results were: precision 0.77, recall 0.85, and F1-score 0.81. The precision was better for the high priority class, but the recall and F1 score were superior in the medium and low priority class, indicating better overall performance.

Dataset	Precision	Recall	F1-score	Accuracy	MCC
GitHub: Medium and low vs. high priority					
Medium and Low	0.77	0.85	0.81	0.80	0.600
High	0.83	0.74	0.79		

Table 5.9: GitHub: medium and low vs. high priority. Inference on the hold out test set.

5.6.2 RQ1.2: Inference on unseen projects

To see how well the models perform in the unseen project, several test sets were used. Three from the GitHub data set: flutter, mypy, and rust. Three from the Jira data set: Qt, MongoDB, and JIRA. This will show if the model can detect high-priority and generalize across projects and platforms.

The results are presented in Table 5.10. The results are varied and none of the test projects performs as good as the hold-out test set.

The best performing test was Qt with an MCC score of 0.418, which is quite surprising considering it comes from the Jira data set. In particular, the recall for the high class was higher than the holdout test with a recall of 0.75 versus 0.74. The other metric was considerably lower than the holdout set, with the rest class dropping from 0.85 in the holdout test set to 0.67 in the Qt data set.

The other Jira data set JIRA and MongoDB performed considerable with MCC score of 0.188 and 0.085 respectively, which is just a bit better than a random score. Also, every other metric dropped considerably.

For the GitHub test project, flutter performed the best with an MCC of 0.347. Rust and mypy performed a bit worse with an MCC of 0.206 and 0.222. Notably, here is that for the high-priority class both flutter and rust had a higher call with 0.84 and 0.83 than the holdout set set. Although mypy had an recall of 0.71 which is a bit low recall than the hold-out test set.

5.6.3 RQ1.3: Fine-tuning on unseen projects

A fine-tuning was performed by including 30% of the unseen project in the GTD data set. This will show if the model can be improved in specific projects. 30% was chosen because Skryseth et al. [25] found this to be the best performing percentage. The goal here is not to find the best possible fine-tuning percentage but to see if there is a considerable difference in performance in the model when it is done. Also, since testing all the incremental training splits is a very resource-intensive task.

All models improve their accuracy and the MCC score a little. The full results can be seen in Table 5.10. In addition, all other metrics seem to improve across the board for most models and classes. In some cases where the pre-fine-tuning metric scored poorly, after fine-tuning it was quite an improvement. Rusts recall of nonhigh priority went from 0.35 to 0.70, but the recall for the high-priority class dropped from 0.83 to 0.61. Although the overall performance of the model increased with an MCC score from 0.206 to 0.309.

Qt from the Jira data set was the best performing model with an increase in MCC from 0.418 to 0.510 and accuracy from 0.71 to 0.75. It shows improvement in both class for all the metrics, notably the highest class the recall increased from 0.75 to 0.83.

Project	Precision	Recall	F1-score	Accuracy	MCC
GitHub holdout test set					
Medium and Low	0.77	0.85	0.81	0.80	0.600
High	0.83	0.74	0.79		
Jira: Qt					
Rest	0.73	0.67	0.70	0.71	0.418
Highest	0.69	0.75	0.72		
Jira: Qt, 30%					
Rest	0.80	0.68	0.73	0.75	0.510
Highest	0.72	0.83	0.77		
Jira: JIRA					
Rest	0.57	0.71	0.63	0.59	0.188
Highest	0.62	0.47	0.54		
Jira: JIRA, 30%					
Rest	0.68	0.64	0.66	0.66	0.345
Highest	0.66	0.70	0.67		
Jira: MongoDB					
Rest	0.55	0.49	0.52	0.54	0.085
Highest	0.54	0.59	0.56		
Jira: MongoDB, 30%					
Rest	0.60	0.56	0.58	0.58	0.181
Highest	0.58	0.62	0.60		
GitHub: Flutter					
Medium and Low	0.75	0.49	0.59	0.66	0.347
High	0.62	0.84	0.71		
GitHub: Flutter, 30%					
Medium and Low	0.76	0.63	0.69	0.72	0.451
High	0.70	0.81	0.75		
GitHub: Rust					
Medium and low	0.67	0.35	0.46	0.59	0.206
Highest	0.56	0.83	0.67		
GitHub: Rust, 30%					
Medium and low	0.63	0.70	0.66	0.65	0.309
High	0.68	0.61	0.64		
GitHub: Mypy					
Rest	0.64	0.51	0.57	0.61	0.222
Highest	0.59	0.71	0.64		
GitHub: Mypy, 30%					
Medium and low	0.67	0.66	0.66	0.66	0.327
High	0.66	0.67	0.67		

Table 5.10: GitHub: medium and low vs high priority. Inference on projects unseen by the model and fine tuning on them with 30% of data from the project included in the training data set.

Chapter 6

Discussion

This section discusses the results of the research questions and presents the models that perform the best. Examines the insights gained from the iterations and their impact on the final artifact. The best models from both the GitHub and Jira datasets will be presented and compared. The implications of the results for practice and research will then be discussed. Following this, the validity and limitations of the research will be addressed. Finally, some final thoughts, related work, and future work will be discussed. Just to make things clearer, when referring to the GitHub model, it is the final artifact found in Section 5.6 and Jira it is Section 5.5.

6.1 Discussion of results of the research questions

Since the main objective is to automatically label high priority issues, the following criteria are especially interesting. First, the sub-research questions will be discussed, then the answers to the main research question Section 1.1

6.1.1 Recall of classified high priority issues

The recall measures the proportion of true positives, identified by the model, from all issues that were high priority in reality, labeled by the developers. The aim is to get a classifier that minimally overlooks high-priority issues, particularly when automatically tracking such issues in a project to analyze its development.

6.1.2 The precision of classified high priority issues, how many were mistakenly classified?

The precision of the model indicates the ratio of true positives (issues identified as high priority by both the model and the developers) out of all issues classified as high priority by the model. The aim is to minimize false positives to reduce the likelihood that a non-high-priority issue is

incorrectly classified as high priority. It is important to note that the set of non-high priority issues might include high priority issues that were not manually labeled. Thing to note here is that issues could have been mislabeled, thus skewing the models ability to identify the classes correctly. But this risk has been reduced by only selecting issues with a priority tag for most of the models.

6.1.3 RQ1.1: How well do Transformers perform for priority classification when the model is trained and tested on the same dataset?

Binary classification is preferable to multi-label

The first iteration shows that binary classification is preferable to multi-label classification for identifying high-priority issues. Binary classification provides better performance metrics than multi-label classification. However, it comes with a trade-off: binary classification can only distinguish between the highest priority and the rest of the issues. This means that it cannot differentiate between medium and low priority. Medium-priority issues, which may involve improving the product with new features or improving existing ones, are still important. However, these issues are not as urgent as high-priority issues that require immediate attention. So, it means that when a company has extra time or all the high priority issue have been dealt with, maybe fixing all issues to polish the product before launch. This approach will not find the medium issues that could have an impact on how well the projects look.

The reason for the better performance of binary classification could be the clearer distinction it provides. When the model is only tasked with distinguishing high-priority issues from all others, it can focus on identifying the most critical characteristics that define high priority. This likely reduces the complexity and ambiguity that comes with trying to classify issues into multiple categories, thereby improving overall performance.

Best Performing GitHub Model: Medium and Low vs. High

For the GitHub model, two binary classification approaches were tested: *medium and low vs. high priority* and *rest vs. high priority*. This was to evaluate how the different approaches would impact the performance of the models.

In *medium and low vs. high*, all issues have a priority tag, ensuring that each issue is classified. Including other issues without a priority tag in the *all-issues* class risks incorporating issues that should have a priority label. For example, some issues might actually be high priority, thus confusing the model. This is reflected in the models performance.

The best performing GitHub model is *medium and low priority vs. high priority*. This model outperforms the distribution of *all issues vs. high*

priority class. The MCC score improved from 0.515 to 0.600, and the accuracy increased from 0.76 to 0.80. Surprisingly, the performance improvement for the recall of high-priority issues was marginal, increasing from 0.73 to 0.74. The precision increased more significantly, from 0.77 to 0.83.

These results are likely due to the *all issues* class, including issues that should be high priority but do not yet have a label. Consequently, high-priority issues might be present in both classes, confusing the model.

Including only issues with assigned priorities ensures that every issue is categorized. Having a priority label on all issues facilitates the models' ability to distinguish between different levels of priority.

The second iteration showed that *all issues vs. high priority* was the best configuration of the class, but this was not the case in the final artifact. This discrepancy could be attributed to the use of a better model, DeBERTaV3, in the final artifact. The superior model might be better at identifying what defines a high-priority issue.

One possible reason for the marginal improvement in recall for high-priority issues could be due to the inherent difficulty in capturing all nuances that define an issue as high-priority. Although precision improved significantly, indicating that the model became better at avoiding false positives, recall did not see as much improvement, suggesting that some high-priority issues still go unrecognized. This might imply that the descriptions of high-priority issues are not always distinct enough from medium- or low-priority issues, making it challenging for the model to identify them based solely on text.

The *medium and low vs. high priority* models are practical for identifying high priority issues. They categorize issues as high priority, while the rest are labeled as nonhigh priority, regardless of whether they are medium or low priority. This approach is not effective if distinguishing between medium- and low-priority is also a goal. Thus, the ability to differentiate between medium and low priority issues is compromised in favor of more accurately identifying high-priority issues.

Future research could focus on enhancing the models ability to distinguish high-priority issues more effectively, possibly by incorporating additional features such as issue metadata or user activity patterns. Furthermore, exploring ways to improve the consistency of the label in the data sets could further enhance the performance of the model.

Best performing Jira model

For the Jira dataset, several models were tested. However, to answer this RQ1.1, the model must be applicable in a practical scenario.

The *lowest vs. highest* model was the best performing, but only included two of the five priority levels in the Jira data set. This shows that the models can

distinguish between priority levels that are far apart. It was also tested *low, lowest vs. highest, high*, which performed a bit worse with the MCC score going from 0.679 to 0.544. The problem with these is that they cannot be used in practice since they leave out priority groups, so the models do not know how to place them. So for a medium issue, it is just guessing which class to put it in. So, a medium issue could end up in both the highest class and the lowest class.

This showed that the best configuration for the Jira dataset was *high, medium, low, and lowest vs. highest*. This setup had the best overall performance and is practical. Attempts were made to remove the shortest 25th percentile of the issues, but this resulted in worse performance, likely because removing issues reduces the model's ability to generalize to other projects.

Removing the shortest 25th percentile of issues might have hindered the model's performance because these shorter issues, while less detailed, still provide valuable context for understanding priority levels. Excluding them could lead to a loss of important data patterns that help the model to generalize better to various projects.

Another approach tested was to switch the high priority level from the rest class to the highest class, making it *medium, low, and lowest vs. highest and high*. The performance was worse than having only the highest as the highest priority class. Moreover, this approach prevents us from distinguishing between the highest- and high-priority issues. Therefore, the Jira model that performed the best was determined to be *high, medium, low, and lowest vs. highest*.

These results suggest that the model performs best when it has a clear distinction between high-priority and lower-priority issues. By including a broader range of priority levels in the rest class, the model likely benefits from a richer dataset that captures more nuanced variations in issue descriptions. This could explain why the *high, medium, low, and lowest vs. highest* configuration performed best overall. Striking a balance between having enough variation to train the model effectively and maintaining clear distinctions that prevent confusion between different priority levels.

The poorer performance of *medium, low, and lowest, vs. highest and high priority* suggests that merging high priority with highest priority issues introduces ambiguity. High-priority issues, while important, may not always share the same critical characteristics as highest-priority issues, making it harder for the model to learn effectively from this combined class. This indicates the importance of maintaining distinct classes for different priority levels to enhance the model's ability to accurately classify issues.

GitHub model performs better than Jira model

For Jira, the model achieved a precision of 0.71 and a recall of 0.72, both of which are moderate. This suggests that the model misses a significant number of high-priority issues.

For GitHub, the performance is slightly better, with a recall of 0.74, which is also moderate, and a precision of 0.83, which is quite good. However, there is still room for improvement in both metrics to ensure more reliable detection of high-priority issues.

The moderate performance of the model in both Jira and GitHub datasets indicates that, while the GitHub model performs better, there is still room for enhancement in detecting high-priority issues. This highlights the need for further refinement and optimization of the models to achieve a more accurate and reliable classification of issue priorities.

These findings suggest that the models benefit from a clear distinction between high-priority and low-priority issues. The better performance of the GitHub model may be due to the dataset's characteristics, such as the quality and consistency of issue descriptions, which help the model learn more effectively. Future research should focus on improving the models' ability to generalize across different datasets and better capture the nuances of high-priority issues.

6.1.4 RQ1.2: How well do Transformers generalize to classify priority in projects unseen by the model?

When transformer models are trained with the GDT dataset and then tested on unseen projects, the results show significant variability depending on the specific projects tested.

Jira model tested on unseen projects by the model

For the Jira final artifact model (Section 5.5), the highest performance was in the Qt project, achieving a precision of 0.75 and a recall of 0.82. This surpassed the results of the holdout test set, which is quite surprising. Overall, performance ranged from moderate to good, but was generally insufficient for practical application in diverse settings. Other test projects produced mixed results. Generally, when the Jira model was tested on Jira projects, it outperformed GitHub projects. However, the MongoDB (Jira dataset) project was one of the lowest performers, with a precision of 0.68 and a recall of 0.58 for the highest priority class. This indicates that many high-priority issues were missed. The Flutter project from GitHub showed a reasonable recall of 0.80 but a lower precision of 0.66.

GitHub models tested on projects unseen by the model

For the GitHub model (Section 5.6), the results were similarly inconsistent.

When the GitHub model was tested on GitHub projects, it generally performed slightly worse than the holdout test set. The best-performing project was Flutter, with a precision of 0.62 and a recall of 0.84 for the high-priority class. This means that the model identifies a high number of high-priority issues. For other projects, the results were a bit worse, with only the recall for the Rust project in the high-priority class standing out at 0.83.

When the GitHub model was tested on Jira projects, the Qt project performed well, but the Jira and MongoDB projects performed very poorly, with the recall for the highest priority class at 0.47 and 0.59, respectively. This indicates that the model missed a considerable number of high-priority issues. The Qt project performed surprisingly well, with a recall of 0.75 for the highest priority class, and other metrics also around 0.70. This indicates that the practices in the Qt project might be more similar to a GitHub project than the other projects tested in Jira.

Jira model generalizes better than the GitHub model

Overall, for testing on unseen projects, the Jira model performs better, although the GitHub model performs better in the holdout test set. This discrepancy can be attributed to the reduced number of issues in the GitHub model by including projects from the top 50 high-priority labels.

This highlights the importance of including a diverse and extensive set of issues in the training data so that transformer models can perform well when applied to a varied set of projects. Reducing the amount of training data, even if it is of higher quality, does not help the model generalize well to other projects.

Ensuring that the training dataset is representative and varied is crucial to achieving better generalization in practical applications.

These findings suggest that the model does not generalize well across different projects, and its applicability depends heavily on the specific context of each project. Therefore, individual testing for each project is necessary if this approach is to be adopted. Given the current results, transformers trained with this model are not recommended for practical use in their current form.

The variability in the performance of the models across different projects could be due to several factors.

Different projects often use domain-specific language and terminology, which might not be well represented in the training data. For instance, technical terms and jargon specific to a project like MongoDB might not be prevalent in other datasets, leading to poorer model performance.

The nature and structure of issues can vary significantly between projects. Projects with more detailed issue descriptions might yield better classifica-

tion results. The surprising performance of the Qt project might be due to well-documented and structured issues that align better with the model's training data.

The criteria for labeling high-priority issues might differ between projects. Some projects may be more stringent in labeling issues as high priority, while others might have a broader interpretation. This inconsistency can affect the model's ability to generalize across different datasets.

The size and diversity of the training dataset play a critical role. A more diverse data set that includes a wide range of projects and issue types is likely to result in better generalization. The reduced performance of the GitHub model on unseen projects might be attributed to a less diverse training set.

The complexity and size of the projects could also influence the performance. Larger and more complex projects might have more nuanced and varied issues, making it harder for the model to generalize from the training data.

6.1.5 RQ1.3: How well do Transformers perform for priority classification when the model is fine-tuned?

Transformers trained on a large GTD dataset and including portions of projects and datasets that were not used during training (fine-tuned) perform a bit better and require a small portion of new high-priority issues included in the training.

Most of the projects saw an increase in most metrics when a 30% was introduced from the test set when training the model. Both the best and worst test sets saw an improvement after fine-tuning.

Fine tuning with GitHub model on unseen projects

For the GitHub model, the best performing test set was Qt, with an MCC of 0.418 to 0.510. Increases in precision indicate that the model returns more relevant results. For the *rest* category, the precision improvement from 0.73 to 0.80 suggests that the model is more accurate in identifying true positives out of all predicted positives. In the highest category, the increase from 0.69 to 0.72 shows a similar trend, albeit more modest. The recall increase from 0.75 to 0.83 in the *highest* class after fine-tuning is particularly significant. This metric reflects the model's ability to find all the relevant cases (true positives). A higher recall means fewer false negatives, which is crucial to ensure critical issues are not overlooked. The F1 score increased from 0.72 to 0.77 for the highest and from 0.70 to 0.73 for the *rest* indicating a better balance between precision and recall, which is beneficial in environments where both are equally important.

The general trend seems to be for the metric that scored poorly on the test set and saw a significant improvement after the fine-tuning. For example,

in the rust project, the rest of the class saw an improvement in recall from 0.35 to 0.70. For the high class many of the projects saw in decrease in the recall for the high class. Rust went from 0.83 to 0.61, flutter from 0.84 to 0.81 and mypy from 0.67 which is not the desired outcome since recall for the high class is the most important metric to consider in this case.

Fine tuning with Jira model on unseen projects

For the Jira model after fine-tuning, the entire project also sees improvements in both the Jira projects and the GitHub projects.

For the Jira model the recall for the highest class stayed around the same or improved a bit in all the projects. Both Qt and Jira saw a one-point drop in both, which is not very significant. With most other metric improving, such as Qt, recall for these rest went from 0.72.

This suggests that fine-tuning is beneficial for most projects. Although the recall for the highest class does not appear to have much improvement in many cases, the other metrics improve. This makes the model better at finding both classes. As it will be more precise on finding highest priority issues and it will find more non-high-priority issues.

The best performing of all the projects was the flutter project from GitHub with the Jira model. The findings of this will be highlighted in the next section as best-case scenario.

The best performing project after fine tuning

Even though the models did not generalize too many projects well, some of them performed pretty well after fine tuning. The best performed were the Jira models in the GitHub flutter project with the MCC going from 0.394 to 0.605 after fine-tuning with 30% of the flutter data.

The Figure 6.1 shows the confusion matrix in the flutter project with the Jira model before fine-tuning. The bottom right and top left quadrants are the field in which we want to have the highest percent.

High, medium, low, lowest priority:

- 29.11% of the issues that were not highest priority were correctly identified as such by the model.
- 20.89% of the issues that were not highest priority were incorrectly identified as high priority by the model.

Highest priority:

- 9.87% of the highest priority issues were incorrectly identified as not high priority by the model.
- 40.13% of the highest priority issues were correctly identified as such by the model.

High, medium, low, lowest priority:

- 39.12% of the issues that were not highest priority were correctly identified as such by the model after fine-tuning.
- 11.15% of the issues that were not highest priority were incorrectly identified as high priority by the model after fine-tuning.

Highest priority:

- 8.62% of the highest priority issues were incorrectly identified as not high priority by the model after fine-tuning.
- 41.11% of the highest priority issues were correctly identified as such by the model after fine-tuning.

From this we see that the true positive(highest priority guessed correct) increase slightly from 40.13% to 41.11% after fine-tuning. It only incorrectly labeled highest priority non highest priority in 8.62 % of the times.

It was a bigger change for the high, medium, low, lowest class were the true negatives (high, medium, low, and lowest correctly predicted) went from 29.11% to 38.12% and the false positives (high, medium, low and lowest predicted as highest priority) 11.1%.



Figure 6.1: Confusion matrix using the Jira model with inference on the flutter project.

So, a project that fits well to the models, it seems that after fine-tuning it shows some promise in identifying high-priority issues. It will still wrongly predict around 10% of labels for each class, but this can be manageable. In this project, all metrics scored reasonably high with around 0.80 across the board. This means that in this case the model identified around 80% for both the high-priority issues and the nonhigh-priority issues.

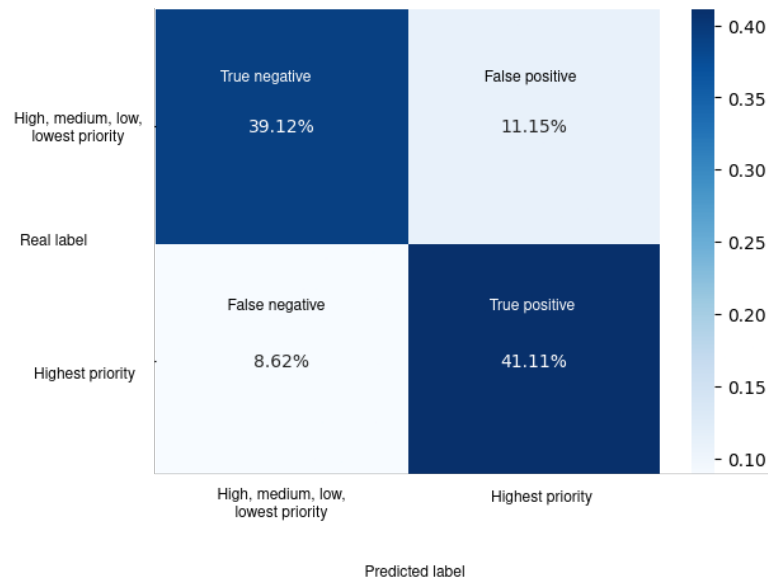


Figure 6.2: Confusion matrix after fine-tuning the Jira model with the flutter project.

6.1.6 RQ1: How well do Transformers support automatic priority classification of issues in issue tracking systems?

After answering the sub-research questions, the following can be said about RQ1. Applying text classification with transformers to the descriptions and titles of issues could help identify high-priority issues in some projects.

Binary classification is the preferred method for transformers in this thesis, comparing high priority to all nonhigh priority issues. This approach is preferable because the multi-label classification performed much worse than the binary classification. However, the drawback is that it cannot distinguish between medium- and low-priority issues.

The transformers' ability to generalize was tested on six different projects that were unseen by the models. The results show that, while the models can perform well on some projects, they struggle with others that do not share the same labeling practices as those most commonly identified by the models.

Fine-tuning of the transformer models improved performance for some projects. However, many projects still suffer from poor performance. The best-case scenario identifies about 80% of all issues for both classes. This is reasonable and could help correctly identify high-priority issues in issue tracking systems.

6.2 Related work

Izadi et al. [14] looked at High-priority vs Low-priority in GitHub issues.

The table below presents a comparison of the results from Izadi et al. [14] and the results from this thesis. The table highlights the metrics for precision, recall, F1 score, accuracy, and MCC in different data sets and evaluation methods.

Dataset	Precision	Recall	F1-score	Accuracy	MCC
Project-based priority prediction results [14]					
Low Priority	0.77	0.78	0.77	0.75	None
High Priority	0.73	0.74	0.72		
Jira model Thesis: Holdout test set					
Lowest Priority	0.86	0.80	0.83	0.84	0.679
Highest Priority	0.82	0.87	0.84		
Cross-project priority prediction results [14]					
Low Priority	0.75	0.83	0.79	0.74	None
High Priority	0.70	0.59	0.64		
Jira model Thesis: Qt project					
Lowest Priority	0.86	0.70	0.77	0.79	0.598
Highest Priority	0.75	0.89	0.81		
Jira model Thesis: MongoDB project					
Lowest Priority	0.76	0.77	0.77	0.76	0.529
Highest Priority	0.77	0.75	0.76		
Jira model Thesis: JIRA project					
Lowest Priority	0.80	0.42	0.55	0.66	0.362
Highest Priority	0.61	0.90	0.72		

Table 6.1: Comparing results from this thesis with results from Izadi et al. [14]

The results indicate that transformer models, when applied to the task of priority classification, outperform the methods used by Izadi et al. [14] in several metrics, particularly in precision and recall for high-priority issues. This demonstrates the potential of transformers to enhance the accuracy and reliability of automatic issue prioritization. Furthermore, the ability of transformers to generalize across different projects, as shown by the evaluations on the Qt, MongoDB, and JIRA projects, underscores their robustness and adaptability to diverse datasets.

The other thing this thesis does differently from other approaches is testing on many unseen projects with the models. In this thesis, both inference and the effect of fine-tuning have been explored.

It also explores a larger GitHub dataset and uses the Jira dataset to cross-check with the models, meaning that Jira models were used to test on projects from GitHub and vice versa. This provided insights into how a model trained on data from one issue tracking system performs on another

system.

Furthermore, it explores how transformers can be applied to all types of priorities, not just bugs, which is a major focus of the literature, as highlighted in Bugayenko et al. [5].

Overall, the combination of extensive cross-system testing, fine-tuning, and binary classification sets the thesis apart from other work in the field.

6.3 Contributions

The following contributions are made by this thesis.

The thesis provides insights into available datasets with priority labels, specifically identifying Jira and GitHub as two primary sources. It also explains how these platforms utilize priority labels, highlighting the under-used of labels on GitHub and the inconsistency in their application.

These data sets have been made into new sub-data sets with issues only having priority labels from both Jira and GitHub. Both cleaned and raw data from the issues are kept to allow others to apply their own pre-processing approaches to the data set.

It has been demonstrated that a binary classification approach yields better results than a multi-label classification approach for classifying priority labels. It also shows that transformers can be used to identify high-priority issues by looking at the text in the title and description of an issue.

It has shown that in a best-case scenario the models give 80% of issues a priority label. And the models work when cross-applied to other issue tracking systems.

The challenges faced by transformer models are highlighted. It shows that transformers have a problem differentiating between priority levels that are closely related, such as high priority vs. medium priority issues.

6.4 Implications

6.4.1 Implications for research

The findings of this thesis have several implications for research within the field of software engineering. Especially prioritization of backlog issues and assigning new issues a priority label in issue tracking systems.

This research expands our understanding of how transformer models can be adapted and optimized for text classification tasks within software development environments, providing a foundation for future innovations in this area.

The evaluation of Transformer-based models in this study, across multiple data sets, including GitHub and Jira, underscores the importance of model

robustness and adaptability. The findings suggest that these models can be effectively transferred across different contexts; for example, a model trained on a Jira dataset can be used effectively on a project from GitHub.

The results demonstrate that some models, such as the one that distinguishes between the lowest and highest priorities in Jira, can accurately label high-priority issues to a significant degree.

The thesis has provided insights into the challenges associated with compiling a robust dataset, where medium-priority issues seem to be particularly difficult for the Transformer model to identify accurately. This challenge requires further research to resolve.

Additionally, the results show that while the approach may be useful for some projects where the outcomes are promising, in other projects the models tested perform very poorly and would not be a good approach to pursue.

When fine tuning on project, all project see an improvement in most metric. Showing that is important to fine-tune the project to increase the performance of the models. The project that performed best when tested in the project unseen by the model also saw an increase in metrics after fine-tuning. Although the project that performed poorly before fine-tuning saw an increase in performance after fine-tuning, most of did not increase enough to be used in practice.

6.4.2 Implications for practice

The practical implication of the findings is that, while Transformer models show potential in automating the classification of priority in issues within software development environments, the current state of models from this thesis seems insufficient for deployment in real-world projects. This conclusion stems from the variability in model accuracy and the specific challenges in correctly classifying medium-priority issues. It may work well in some project after fine-tuning.

Although the current models in this thesis are not ready for widespread deployment, there may be specific conditions under which they could still provide value. In a best-case scenario where the model correctly predicts around 80% of the issues, it could potentially be used in a practical setting. To achieve this, a prototype needs to be created and the feasibility of the approach studied. This is necessary to determine if the suggested approach saves developers time by assisting in labeling high-priority issues, or if it adds more confusion by misidentifying non-high-priority issues as high priority. It is also crucial to ensure that critical issues are not missed by the model labeling high-priority issues as non-high priority.

For example, they might be used as a supplementary tool for issue tracking systems, where they can pre-filter or pre-prioritize issues before human review. This could streamline the workflow and reduce the workload for

human operators, provided that the limitations of the model are clearly understood and managed. This is something that needs to be further explored if such an approach would save time or just add confusion.

6.5 Quality of research and limitations

This section examines the quality of the research on automatic issue prioritization. Based on the validity classifications of Wohlin et al. [31], the construct validity, the internal validity, the external validity, the validity of the conclusion and the reliability are discussed in the context of the study.

6.5.1 Validity and Validation

Validity refers to the degree to which an inference, statement, or assertion is considered accurate or correct Wohlin et al. [31], ideally supported by evidence or data. It indicates the trustworthiness of the results and makes sure that the results are as objective as possible and not tainted by the views of the researchers. Validation involves examining the validity of a particular claim Borsboom et al. [3].

6.5.2 Construct validity threat

Construct validity is crucial to ensure that the conclusions drawn from the research are based on reliable and relevant measurements [31].

One limitation and threat to the validity of the work is that the performance of the model is at the mercy of the priority labels set on the issues. In issue trackers such as GitHub and Jira, the priority level of issues is typically assigned by project managers, product owners, or team leads. These roles are responsible for overseeing the project's progress and ensuring that resources are allocated effectively. They set priorities based on the impact of the issue on the project, its urgency and the available resources. Developers and other team members can suggest priority levels based on their understanding of the issue's technical implications and the overall project needs, but final prioritization is usually confirmed by the leadership or those in project management roles. This helps in managing the workflow and addressing the most critical issues first to streamline development and meet project milestones efficiently.

So, the people who set the priority levels naturally have good knowledge about the project and the domain, so overall the priority labels set should be good quality. In the GitHub models to select good quality data things like only using the repositories with the most high-priority issues, to make sure that the model is trained to the highest possible data as possible.

Even if you have strong domain knowledge and know a lot of the project, it can be hard to differentiate between the different priority levels, especially medium issues. The Jira model showed good results when only trained on the highest vs. the lowest priority issues, and such identifying medium as the the hardest priority level to label correctly.

6.5.3 Internal validity

Internal validity is critical in empirical research to establish that observed effects can be directly attributed to the hypothesized cause of the researcher, rather than external factors [31]. In the context of this thesis, ensuring internal validity involves demonstrating that the predictive success of these models is genuinely due to their algorithmic capabilities and not influenced by confounding variables.

One potential threat to internal validity is selection bias in the data set and the projects used to train the models. Efforts were made to create as varied a data set as possible and include many types of projects. In the Jira data set, most of the projects were included. Only a few were excluded because their number of priority issues was too small.

Changes in how data are collected, processed, or used during research could lead to inconsistencies and pose a threat to internal validity. In this thesis, all preprocessing and cleaning techniques were consistently applied to both the training and test data sets. Therefore, there is little chance that this would impact the results of the models.

Overfitting could also threaten internal validity. To address this, a 3-fold cross-validation was employed to assess the models' performance. This method not only helps in validating the models effectiveness across different subsets of data but also prevents overfitting by ensuring that the model can perform well on unseen data.

Another internal validity threat could be that the selection of features in the thesis might not capture all the nuances that human project managers consider when prioritizing issues. In this thesis, only text from titles and descriptions of the issues is used to train the transformers. Overlooking features such as the number of comments, user impact, or historical data on similar issues could limit the models real-world applicability.

However, this exclusion was intentional, since the main focus of the thesis is to assess how well transformers perform based solely on the descriptions and title of the issues. However, incorporating more features at a later stage is a possibility that could be explored to see if it enhances the performance of the models. But some of those features, the issues must be unlabeled for a time. For example, the number of comments is not very relevant to newly created issues, since it does not have comments. One of the goals of the thesis is to give a priority label to an issue when it is created. This approach would only work on a backlog of issues with no priority label.

6.5.4 External validity

External validity looks at how well the results can generalize [31]. Threats to external validity refer to conditions that limit the ability of the results to be generalized. The external validity of this research refers to the way in which the models generalize from the GitHub and Jira contexts to other settings, platforms, and types of software development projects. This

aspect of validity is crucial for assessing whether the models developed can be effectively generalized across various real-world environments.

It is possible that the trained models do not generalize well to other platforms such as GitLab, Bugzilla, and Bitbucket. One step to mitigate this in the thesis is that the models were cross-applied to other platforms. Jira models were tested on GitHub projects and vice versa. The results show that in some cases when the model is cross-applied, it performs well, whereas in other projects it does not. If other issue tracking systems are not similar to Jira and GitHub, it could result in poor performance by the model in these environments. GitHub and Jira are the most used issue tracking systems, and thus should cover a significant portion of software projects. However, if the approach were to be used in other platforms, the approach may need to be refined.

External validity also depends on the ability of the models to handle a wide range of problem types, from simple bug reports to complex feature requests. The current study used only public repositories, which might not fully capture the complexity or diversity of issues managed in large-scale commercial software development. Therefore, the model might not perform well in closed source environments.

6.5.5 Conclusion validity

Conclusion validity refers to the degree to which conclusions reached about relationships in the data are reasonable [31].

The validity of the conclusion is essential to determine in a positive way whether the relationships observed in our study are statistically significant and not due to random errors or inappropriate analysis methods. In the context of this thesis, ensuring conclusion validity is crucial for underpinning the reliability of our empirical findings.

Throughout this research, machine learning evaluation metrics were used to assess the performance of Transformer models across GitHub and Jira data sets. Ensuring the validity of our conclusions required meticulous application of these methods. We used standard metrics such as accuracy, precision, recall, and the F1 score to evaluate model performance, each chosen for their relevance to classification tasks and their widespread acceptance in machine learning research.

Care was taken to ensure that these metrics were calculated in a manner consistent with best practices in the field, including the use of cross-validation techniques to mitigate overfitting and provide a robust estimate of model performance across different data subsets. This approach enhances the reliability of our conclusions, ensuring that they are not merely artifacts of peculiarities in a specific data set or a particular split of training and testing data.

6.5.6 Limitations

This study has several limitations that should be acknowledged and addressed in future research. Some of the key limitations include the following.

Dataset quality and availability: The quality and availability of Jira and GitHub data sets may affect model performance. The data sets used in this study were limited, and future studies should aim to collect and incorporate more diverse and high-quality data sets.

It was also used only open source data sets where both regular people and people in tech can make an issue. So, it could be that using closed source data set could be better if the models were to be used on closed source issue tracking systems. But one of the problems these models are trying to deal with is the high amount of issues created by anyone on GitHub or Jira.

In an open source project anyone can make an issue, so the data is a mix of regular people making issues and people working in tech.

Class imbalance: This thesis uses a fairly simple way to deal with class imbalance, random undersampling of the majority class. This was a choice made to keep things simple and allowed for more time to do different experiments in different iterations. Other more powerful techniques could be explored to see if it would affect the performance of the models. The data cleaning process was fairly simple. If you want to maximize the performance of a model in production, the data cleaning must be examined in more detail. This can help determine whether it improves the performance of the models.

Limited scope of issues: This thesis only looks at finding high-priority issues and cannot find the difference between medium and low, which is still useful. So efforts could be made to make models that deal with this problem.

6.6 Final thoughts

This thesis has explored the application of transformers to automatic classification of priority in issues by giving them a specific priority label in software development projects. The results of this study demonstrate the potential of transformers in identifying high-priority issues, but also highlight the limitations and challenges of applying these models in real-world projects.

One of the main takeaways is that it really depends on what projects the models are used on. The results suggest that fine-tuning and adaptation to specific projects are crucial to achieving reliable and accurate results.

This thesis contributes to research on automating the prioritization of issues in issue tracking systems. It highlights the need for further research and

development in this area. Future studies can build on the findings of this study, exploring new approaches and techniques to improve the accuracy and robustness of issue prioritization models.

The goal of automated issue prioritization is to support software development teams in managing and prioritizing issues more efficiently and effectively. This study demonstrates the potential of transformers in achieving this goal in some projects. But is currently not good enough to be offered as a solution in many projects. The need for continued research and development to overcome the challenges and limitations of these models is emphasized.

6.7 Future work

There are several things that could be investigated to improve the performance of the priority models.

Currently, models are only trained with data from one source per model. It could be interesting to see if blending data from both GitHub and Jira would affect the performance and to see if it generalizes well to other platforms.

The thesis only looks at finding the highest priority, so finding method to find the difference between all the priority levels so that medium- and low-priority issues could also be identified.

It could also be interesting if a similar approach as was used on closed-source data to see how that would change the performance of the models.

When the performance of the models to label priority is good enough, it will be interesting to see how it performs in a real scenario and compare it to labeling from humans.

Chapter 7

Conclusion

This thesis explored the use of transformer models for automatic prioritization of issues in issue tracking systems, focusing on the efficiency and precision of priority classification. The study used a design science research approach, applying models such as DistilBERT and DeBERTaV3 to GitHub and Jira datasets.

The research involved three main iterations. Initially, the model was trained to classify issues as high, medium, and low priority. The results were suboptimal, with accuracy barely exceeding random guessing. The high priority class achieved low precision and recall, indicating that multilabel classification was not effective for this task.

The focus then shifted to binary classification, comparing different configurations such as high vs. rest and high vs. medium and low. Binary classification significantly improved the performance of the models, particularly in distinguishing high-priority issues from others. The high priority vs. all issues model achieved an accuracy of 0.70, with better precision and recall than the multilabel approach.

Further iterations involved improving the quality of the data by filtering the issues according to length and selecting repositories with a substantial number of high-priority issues. These improvements resulted in a marked increase in model performance.

The final artifact, developed on the basis of insights from these iterations, demonstrated that transformer models could effectively automate issue prioritization under certain conditions. However, the variability in model performance across different datasets highlighted the need for further refinement and context-specific tuning.

In conclusion, while transformer models show promise for automating issue prioritization, the models in this thesis are not yet ready for deployment in real-world projects. The models could serve as supplementary tools, pre-filtering issues before human review, streamlining workflows, and reducing the workload for project managers and developers. Future

work should focus on improving the generalizability and robustness of the model across diverse project environments.

Bibliography

- [1] Ethem Alpaydin. *Introduction to Machine Learning, fourth edition*. Google-Books-ID: tZnSDwAAQBAJ. MIT Press, 24th Mar. 2020. 709 pp. ISBN: 978-0-262-04379-3.
- [2] Emily M. Bender et al. ‘On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?’ In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. FAccT ’21. New York, NY, USA: Association for Computing Machinery, 1st Mar. 2021, pp. 610–623. ISBN: 978-1-4503-8309-7. DOI: 10.1145/3442188.3445922. URL: <https://dl.acm.org/doi/10.1145/3442188.3445922> (visited on 07/04/2024).
- [3] Denny Borsboom, Gideon Mellenbergh and Jaap Heerden. ‘The Concept of Validity’. In: *Psychological review* 111 (1st Oct. 2004), pp. 1061–71. DOI: 10.1037/0033-295X.111.4.1061.
- [4] Jan vom Brocke, Alan Hevner and Alexander Maedche. ‘Introduction to Design Science Research’. In: *Design Science Research. Cases*. Ed. by Jan vom Brocke, Alan Hevner and Alexander Maedche. Progress in IS. Cham: Springer International Publishing, 2020, pp. 1–13. ISBN: 978-3-030-46781-4. DOI: 10.1007/978-3-030-46781-4_1. URL: https://doi.org/10.1007/978-3-030-46781-4_1 (visited on 08/11/2023).
- [5] Yegor Bugayenko et al. ‘Prioritizing tasks in software development: A systematic literature review’. In: *PLOS ONE* 18.4 (6th Apr. 2023). Publisher: Public Library of Science, e0283838. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0283838. URL: <https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0283838> (visited on 10/04/2024).
- [6] Anton Chernyavskiy, Dmitry Ilvovsky and Preslav Nakov. ‘Transformers: “The End of History” for Natural Language Processing?’ In: *Machine Learning and Knowledge Discovery in Databases. Research Track*. Ed. by Nuria Oliver et al. Cham: Springer International Publishing, 2021, pp. 677–693. ISBN: 978-3-030-86523-8. DOI: 10.1007/978-3-030-86523-8_41.
- [7] Davide Chicco and Giuseppe Jurman. ‘The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation’. In: *BMC Genomics* 21.1 (Dec. 2020). Number: 1 Publisher: BioMed Central, pp. 1–13. ISSN: 1471-2164. DOI: 10.1186/s12864-019-6413-7. URL: <https://bmcbgenomics.biomedcentral.com/articles/10.1186/s12864-019-6413-7> (visited on 13/04/2024).

- [8] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 24th May 2019. DOI: 10.48550/arXiv.1810.04805. arXiv: 1810.04805[cs]. URL: <http://arxiv.org/abs/1810.04805> (visited on 08/04/2024).
- [9] Thi Thu Hang Do, Markus Dobler and Niklas Kühl. *What to Prioritize? Natural Language Processing for the Development of a Modern Bug Tracking Solution in Hardware Development*. 28th Sept. 2021. DOI: 10.48550/arXiv.2109.13825. arXiv: 2109.13825[cs]. URL: <http://arxiv.org/abs/2109.13825> (visited on 27/01/2024).
- [10] Trevor Hastie, Robert Tibshirani and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY: Springer, 2009. ISBN: 978-0-387-84857-0 978-0-387-84858-7. DOI: 10.1007/978-0-387-84858-7. URL: <http://link.springer.com/10.1007/978-0-387-84858-7> (visited on 08/04/2024).
- [11] Pengcheng He, Jianfeng Gao and Weizhu Chen. *DeBERTaV3: Improving DeBERTa using ELECTRA-Style Pre-Training with Gradient-Disentangled Embedding Sharing*. 24th Mar. 2023. DOI: 10.48550/arXiv.2111.09543. arXiv: 2111.09543[cs]. URL: <http://arxiv.org/abs/2111.09543> (visited on 11/04/2024).
- [12] Yingying He et al. 'Understanding and Enhancing Issue Prioritization in GitHub'. In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). ISSN: 2643-1572. Sept. 2023, pp. 813–824. DOI: 10.1109/ASE56229.2023.00044. URL: <https://ieeexplore.ieee.org/document/10298431> (visited on 10/04/2024).
- [13] Jeff Heaton. 'Ian Goodfellow, Yoshua Bengio, and Aaron Courville: Deep learning'. In: *Genetic Programming and Evolvable Machines* 19.1 (1st June 2018), pp. 305–307. ISSN: 1573-7632. DOI: 10.1007/s10710-017-9314-z. URL: <https://doi.org/10.1007/s10710-017-9314-z> (visited on 08/04/2024).
- [14] Maliheh Izadi, Kiana Akbari and Abbas Heydarnoori. *Predicting the Objective and Priority of Issue Reports in Software Repositories*. 28th Sept. 2021. DOI: 10.48550/arXiv.2012.10951. arXiv: 2012.10951[cs]. URL: <http://arxiv.org/abs/2012.10951> (visited on 03/04/2024).
- [15] Gareth James et al. *An Introduction to Statistical Learning*. Vol. 103. Springer Texts in Statistics. New York, NY: Springer, 2013. ISBN: 978-1-4614-7137-0 978-1-4614-7138-7. DOI: 10.1007/978-1-4614-7138-7. URL: <http://link.springer.com/10.1007/978-1-4614-7138-7> (visited on 08/04/2024).
- [16] Dan Jurafsky and James H. Martin. *Speech and Language Processing*. 4th Feb. 2024. URL: <https://web.stanford.edu/~jurafsky/slp3/> (visited on 08/04/2024).

- [17] Cansu Kandemir and Holly AH Handley. 'Employee-task assignments for organization modeling: a review of models and applications'. In: (2014). URL: https://digitalcommons.odu.edu/emse_fac_pubs/158/ (visited on 18/05/2024).
- [18] Miroslav Kubat and Stan Matwin. 'Addressing the curse of imbalanced training sets: one-sided selection'. In: *Icml*. Vol. 97. Issue: 1. Citeseer, 1997, p. 179. URL: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=e3c3914181d76c817f0e35f788b7c4c0f80abb07> (visited on 02/05/2024).
- [19] Tsung-Yi Lin et al. 'Focal Loss for Dense Object Detection'. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017 IEEE International Conference on Computer Vision (ICCV). ISSN: 2380-7504. Oct. 2017, pp. 2999–3007. DOI: 10.1109/ICCV.2017.324. URL: <https://ieeexplore.ieee.org/document/8237586> (visited on 08/04/2024).
- [20] Lloyd Montgomery, Clara Lüders and Walid Maalej. 'An Alternative Issue Tracking Dataset of Public Jira Repositories'. In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 23rd May 2022, pp. 73–77. DOI: 10.1145/3524842.3528486. arXiv: 2201.08368[cs]. URL: <http://arxiv.org/abs/2201.08368> (visited on 11/03/2024).
- [21] Ken Peppers et al. 'A Design Science Research Methodology for Information Systems Research'. In: *Journal of Management Information Systems* 24.3 (1st Dec. 2007). Publisher: Routledge _eprint: <https://doi.org/10.2753/MIS0742-1222240302>, pp. 45–77. ISSN: 0742-1222. DOI: 10.2753/MIS0742-1222240302. URL: <https://doi.org/10.2753/MIS0742-1222240302> (visited on 09/11/2023).
- [22] N. Ur-Rahman and J. A. Harding. 'Textual data mining for industrial knowledge management and text classification: A business oriented approach'. In: *Expert Systems with Applications* 39.5 (1st Apr. 2012), pp. 4729–4739. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2011.09.124. URL: <https://www.sciencedirect.com/science/article/pii/S0957417411014473> (visited on 08/04/2024).
- [23] Denis Rothman. *Transformers for Natural Language Processing and Computer Vision - Third Edition*. 2024. ISBN: 978-1-80512-872-4. URL: <https://learning.oreilly.com/library/view/transformers-for-natural/9781805128724/> (visited on 11/04/2024).
- [24] Victor Sanh et al. *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. 29th Feb. 2020. DOI: 10.48550/arXiv.1910.01108. arXiv: 1910.01108[cs]. URL: <http://arxiv.org/abs/1910.01108> (visited on 17/03/2024).
- [25] Daniel Skryseth et al. 'Technical Debt Classification in Issue Trackers using Natural Language Processing based on Transformers'. In: *2023 ACM/IEEE International Conference on Technical Debt (TechDebt)*. 2023 ACM/IEEE International Conference on Technical Debt (TechDebt). Melbourne, Australia: IEEE, May 2023, pp. 92–101. ISBN:

9798350311945. DOI: 10.1109/TechDebt59074.2023.00017. URL: <https://ieeexplore.ieee.org/document/10207085/> (visited on 05/03/2024).
- [26] Emma Strubell, Ananya Ganesh and Andrew McCallum. *Energy and Policy Considerations for Deep Learning in NLP*. 5th June 2019. DOI: 10.48550/arXiv.1906.02243. arXiv: 1906.02243[cs]. URL: <http://arxiv.org/abs/1906.02243> (visited on 08/04/2024).
- [27] *Summary of the tokenizers*. URL: https://huggingface.co/docs/transformers/tokenizer_summary (visited on 29/03/2024).
- [28] *Tokenizer*. URL: https://huggingface.co/docs/transformers/en/main_classes/tokenizer (visited on 07/05/2024).
- [29] Ashish Vaswani et al. *Attention Is All You Need*. 1st Aug. 2023. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762[cs]. URL: <http://arxiv.org/abs/1706.03762> (visited on 08/04/2024).
- [30] *What are priority levels in Jira Service Management?* | Jira Service Management Cloud. Atlassian Support. URL: <https://support.atlassian.com/jira-service-management-cloud/docs/what-are-priority-levels-in-jira-service-management/> (visited on 24/04/2024).
- [31] Claes Wohlin et al. *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer, 2012. ISBN: 978-3-642-29043-5 978-3-642-29044-2. DOI: 10.1007/978-3-642-29044-2. URL: <http://link.springer.com/10.1007/978-3-642-29044-2> (visited on 05/04/2024).
- [32] Thomas Wolf et al. *HuggingFace's Transformers: State-of-the-art Natural Language Processing*. 13th July 2020. DOI: 10.48550/arXiv.1910.03771. arXiv: 1910.03771[cs]. URL: <http://arxiv.org/abs/1910.03771> (visited on 08/04/2024).
- [33] Syed Farhan Alam Zaidi, Honguk Woo and Chan-Gun Lee. 'Toward an Effective Bug Triage System Using Transformers to Add New Developers'. In: *Journal of Sensors* 2022 (8th Apr. 2022). Publisher: Hindawi, e4347004. ISSN: 1687-725X. DOI: 10.1155/2022/4347004. URL: <https://www.hindawi.com/journals/js/2022/4347004/> (visited on 03/04/2024).

Chapter 8

Appendix

8.1 Code

Scripts used to pre-process, clean and train the models can be found in this GitHub repository: https://github.com/Kristmh/thesis_code

8.2 Use of AI

In this scientific work, generative artificial intelligence (AI) has been used. All data and personal information have been processed in accordance with the University of Oslo's regulations, and I, as the author of the document, take full responsibility for its content, claims, and references. An overview of the use of generative AI is provided below.

The writefull service ¹ has been used to improve the writing of the thesis. It suggests improvements to the writing of the text but does not alter the content of the text.

The tool GitHub Copilot, created by Microsoft, ² has been used to help write and debug code. The tool ChatGPT, created by OpenAi, ³ has been used to debug and understand code.

¹<https://www.writefull.com/>

²<https://github.com/features/copilot>

³<https://chatgpt.com/>