A LEXICON-BASED APPROACH FOR SOFTWARE BUG SEVERITY CLASSIFICATION

Fatima Aziz

Master Thesis
Jožef Stefan International Postgraduate School
Ljubljana, Slovenia

Supervisor: Asst. Prof. Dr. Martin Žnidaršič, Jožef Stefan International Postgraduate School and Jožef Stefan Institute, Ljubljana, Slovenia

Evaluation Board:
Asst. Prof. Dr. Senja Pollak, Chair, Jožef Stefan International Postgraduate School and Jožef Stefan Institute, Ljubljana, Slovenia
Asst. Prof. Dr. Tea Tušar, Member, Jožef Stefan International Postgraduate School and Jožef Stefan Institute, Ljubljana, Slovenia
Asst. Prof. Dr. Martin Žnidaršič, Member, Jožef Stefan International Postgraduate School and Jožef Stefan Institute, Ljubljana, Slovenia

MEDNARODNA PODIPLOMSKA ŠOLA JOŽEFA STEFANA
JOŽEF STEFAN INTERNATIONAL POSTGRADUATE SCHOOL

Fatima Aziz

A LEXICON-BASED APPROACH FOR SOFTWARE BUG SEVERITY CLASSIFICATION

Master Thesis

PRISTOP ZA RAZVRŠČANJE RESNOSTI PROGRAMSKIH NAPAK NA PODLAGI LEKSIKONOV

Magistrsko delo

Supervisor: Asst. Prof. Dr. Martin Žnidaršič

Ljubljana, Slovenia, February 2025

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Asst. Prof. Dr. Martin Žnidaršič, for his continuous support, patience, and motivation during my research. His guidance was invaluable in all stages of this work. I am also grateful to Slovenian Research Agency for research core funding for the programme Knowledge Technologies for partially providing the financial support through grant number [P2-0103]. Without their generosity, this research would not have been possible.

Abstract

Bug severity classification is a critical and time-consuming aspect of the software bug resolution process, as the severity of a reported bug influences its urgency to fix it. This task is often automated using supervised machine learning methods; however, there is a notable lack of dedicated lexicons for this purpose. In this thesis, we assessed the potential usefulness of developing such a resource, implemented it and compared its performance with classic machine learning approaches and general purpose lexicons used in text classification. We also proposed and tested a novel lexicon development approach based in the domain of software bug reports. In our empirical assessment, we used publicly available datasets of bug reports for Firefox and Eclipse. The results indicate that our lexicon approach achieved a comparable F1-score to classic machine learning models. Our experiments revealed that lexicon-based approaches were effective in most of the experiments, while machine learning methods performed better in the experiment that had a more balanced data distribution. The findings highlighted the dataset-dependent nature of classification performance and also indicated the unexpected usefulness of general sentiment analysis lexicons. These results suggest that tailored lexicon-based approaches are a valuable alternative to machine learning techniques for bug severity classification and could potentially reduce the need for large labeled training sets.

Povzetek

Razvrščanje napak v programski opremi glede na njihovo resnost je pomemben vidik odpravljanja napak programske opreme, saj resnost prijavljene napake vpliva na nujnost in posledično potek njene odprave. To razvrščanje je pogosto samodejno, rešeno z uporabo metod nadzorovanega strojnega učenja. Opazili pa smo pomanjkanje namenskih leksikonov in na leksikonih temelječih metod za to nalogo. V tem delu smo ocenili potencialno uporabnost takih leksikonov, jih nekaj razvili in primerjali njihovo učinkovitost z učinkovitostjo klasičnih pristopov strojnega učenja in z leksikoni, ki so bili ustvarjeni za sorodne namene. Pri tem smo predlagali in preizkusili tudi nov pristop za razvoj leksikona. V našem empiričnem vrednotenju smo uporabili javno dostopne nabore podatkov o programskih napakah za orodji Firefox in Eclipse. Rezultati naše študije kažejo, da je naš leksikonski pristop dosegel primerljivo oceno F1 kot klasični pristopi strojnega učenja. Ugotavljamo tudi, da so pristopi, ki temeljijo na leksikonih, učinkoviti v večini poskusov, medtem ko so se metode strojnega učenja izkazale predvsem v poskusu, pri katerem je bil uporabljen nabor podatkov z relativno uravnoteženo porazdelitvijo ciljnih razredov. Izsledki poskusov nakazujejo odvisnost rezultatov od nabora podatkov, pa tudi nepričakovano uporabnost leksikonov, ki so bili ustvarjeni za analiziranje razpoloženja. V splošnem naši rezultati kažejo, da so pristopi, ki temeljijo na leksikonih, lahko alternativa tehnikam strojnega učenja v domeni razvrščanja napak v programski opremi in jih lahko uporabimo za zmanjšanje potreb po označenih podatkih.

Contents

List of Figures

List of Tables

Abbreviations

BERT	...	Bidirectional Encoder Representations from Transformers
CNN	...	Convolutional Neural Network
ELM	...	Extreme Learning Machine
FP	...	False Positive
FN	...	False Negative
ITIS	...	Information Technology and Information Systems
JSON	...	JavaScript Object Notation
LLM	...	Large Language Model
LSVM	...	Linear Support Vector Machine
LogReg	...	Logistic Regression
ML	...	Machine Learning
MultiNB	...	Multinomial Naive Bayes
NLTK	...	Natural Language Toolkit
NLP	...	Natural Language Processing
RNN	...	Recurrent Neural Network
SO	...	Sentiment Orientations
SVM	...	Support Vector Machine
THR	...	Threshold
TN	...	True Negative
TP	...	True Positive
VADER	...	Valence Aware Dictionary and sEntiment Reasoner

Glossary

- **Bing Liu Lexicon (lexicon_BL)**: Bing Liu's opinion lexicon.
- **CodeBERT**: A pre-trained model developed by Microsoft for understanding both programming languages (PL) and natural languages (NL).
- **CR-SMOTE**: A synthetic minority over-sampling technique.
- **Dynamic Lexicon**: A lexicon generated on the fly and adjusted in real time based on a varying combination of thresholds.
- **Lexicon**: A lexicon refers to a collection of words and phrases curated specifically for the textual analysis for the classification of software bug severity.
- **Linear SVM-Based Approach Lexicon (lexicon_SVM)**: A lexicon created with a Linear SVM-based approach on a cross dataset.
- **NEW**: The status of a newly reported bug.
- **Non-Severe Ratio**: The ratio of words in non-severe cases.
- **Non-Severe Threshold**: The threshold defined for non-severe cases.
- **RESOLVED**: The status of a resolved bug.
- **Severe Ratio**: The ratio of words in severe cases.
- **Severe Threshold**: The threshold defined for severe cases.
- **Static Lexicon**: A lexicon stored in a JSON file that remains unchanged.
- **Threshold-Based Approach Lexicon (lexicon_THR)**: A lexicon created with a threshold-based approach using a cross dataset.
- **SEV_THR_LEX_c**: A lexicon created with a threshold-based approach on the combined dataset of Firefox and Eclipse.
- **SEV_THR_LEX_f**: A lexicon created with a threshold-based approach on the combined dataset of Eclipse and Firefox but filtered with the bug status "Normal".
- **VERIFIED**: The status of a verified bug.
- **WONTFIX**: The status of a bug that will not be fixed.

Chapter 1

Introduction

Our lives are increasingly affected by software systems, and effectively managing these systems is becoming more complex and relevant. Given this complexity, the prevention, detection, and correction of software errors are among the most critical activities in ensuring the reliability and functionality of these systems. Software bugs are defined as faults that hinder the intended operation of a software system. Bugs are reported to bug-tracking systems like Bugzilla and Jira in the form of a bug report, which can vary from minor issues that have minimal impact on functionality to critical problems that may cause the software to crash. A bug report usually includes a concise description of the underlying issue, which is important for development teams to understand and allows them to reproduce the issue before attempting a fix. However, the state of these descriptions can vary depending on the reporter's expertise; experienced testers may provide more detailed reports than non-experienced ones. As a result, the development team usually requires additional information before they can address the reported issues correctly [1].

Proper identification and resolution of bugs are crucial for ensuring software functionality, quality, dependability, and a satisfactory user experience. Prioritizing bugs based on their severity allows developers and testers to concentrate on high-impact issues first, leading to quicker resolutions of critical problems and enhancing user satisfaction [2]. Addressing high-severity bugs accordingly not only improves overall software reliability but also facilitates efficient allocation of development resources, which is vital for successful product rollouts and future growth [3]. Prioritizing the most critical bugs allows organizations to streamline their development processes and consistently deliver high-quality software. Efficient bug triaging relies heavily on the textual descriptions found in bug reports. The initial step of bug triaging involves evaluating a bug's impact on various aspects such as software functionality, data integrity, and user experience. This thorough analysis is essential for determining the true impact of a bug, which then allows for the assignment of its severity level. Common classifications of severity levels of bugs include annotations: critical, high, medium, and low. Misclassification of a severity level, particularly its overestimation, can lead to the misallocation of resources where minor issues receive undue attention at the expense of more critical problems. Therefore, accurate classification fosters better collaboration and clarity within the development process, ensuring timely delivery of good quality software while satisfying the needs of users [4].

Traditional methods for the identification and classification of software bugs usually require manual examination, which can be time-consuming and prone to errors, especially in large-scale projects. To address these challenges, several studies have explored machine learning, deep learning, and sentiment-lexicon-based approaches for automating this classification process. Some of the commonly used machine learning models include Support Vector Machines (SVM), Decision Trees, and Random Forests, which were trained on historical bug report datasets to classify the severity of software bugs [5]. Commonly used deep learning techniques include Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), due to their ability to capture complex relationships in textual data [2]. Additionally, sentiment analysis

has been employed in bug classification systems by using sentiment lexicons, which assess the emotional tone of bug reports. This provides insights into the urgency and potential impact of issues [6]. Interestingly though, domain-specific lexicon approaches were very rarely used for this task and there does not yet exist a publicly available resource of this kind.

1.1 Problem Definition and Motivation

The current methods for bug severity classification primarily rely on machine learning models and sentiment-based analysis, which may not sufficiently capture the true impact of a software bug. Lexicon-based approaches address this limitation by providing a structured set of terms and relationships to assess software bug severity. These approaches leverage domain-specific terminology and context, offering a promising and transparent alternative for accurately predicting software bug severity. However, the lack of a freely available and well-curated lexicon for this classification remains a significant gap in the existing research.

The motivation for the current work is as follows:

- Predicting software bug severity is crucial for prioritizing bug fixes in software development.
- Most lexicon-based approaches for bug triaging rely on sentiment analysis, while problem specific approaches have not been commonly used for this task.
- Lexicon-based methods provide an effective way to classify bugs based on predefined keywords associated with severity.
- A well-curated general lexicon can serve as a source of seed words for machine learning and word embedding techniques.
- Lexicons can act as baseline solutions when data or computational resources are limited.
- A lexicon-based approach has the potential to enhance the accuracy, efficiency, and transparency of software bug severity prediction.

1.2 Objectives

The objective of this work is to assess the potential of a lexicon-based approach for bug severity classification. To this end, the aim is to inspect whether lexicon-based approaches can be used as relevant baseline solutions for the development of a robust general lexicon of this kind. Furthermore, this work aims to develop a comprehensive and freely available static lexicon and compare its performance against existing lexicons.

This thesis aims to answer the following research questions.

- Can a lexicon-based approach be a competitive alternative to classic machine learning approaches for software bug severity prediction?
- What is the efficiency of the lexicon-based approach in terms of time consumption compared to other methods?
- How does our developed static lexicon perform compared to existing sentiment analysis-based lexicons used in this problem domain?

1.3 Contributions

The expected scientific contributions of the proposed work are as follows:

- Design and development of a freely available general lexicon for software bug severity classification.
- Empirical evaluation of the effectiveness of lexicon-based approaches compared to traditional machine learning methods and existing sentiment-based lexicons.

1.4 Organization of the Thesis

The rest of this thesis is structured as follows. Chapter 2 discusses the background and related work. Chapter 3 covers data source and data preparation Chapter 4 explains the methodology. Chapter 5 details the experimental settings of our research. Chapter 6 presents the results and discussion that will also include the qualitative analysis of the static lexicons. Chapter 7 provides the source code and static lexicon in JSON format. Finally, Chapter 8 concludes the thesis.

Chapter 2

Background and Related Work

Classification of software bug severity has become an important research area, especially within large-scale open-source projects like Eclipse and Firefox. This process is important for proper bug triaging, allowing developers to prioritize their tasks and focus on the most critical issues [7]. This process is an important part of software maintenance in the software development cycle. To speed up the process of bug triaging, bugs need to be accurately classified and assigned to appropriate developers. Since software testing and bug triaging are time-consuming activities, proper management of bugs is important in minimizing the cost of the software development process. A general bug report contains various attributes such as Bug ID, Status, Submission Date, Summary, Severity, Description, and Priority [8]. Based on the information in the summary or description in the bug reports, the bug reporters usually classify the bug according to its severity [9] which affects how the bug is prioritized and assigned to the related developer [10]. This process of assigning a severity level to the bug is error-prone highlighting a significant gap in the bug severity classification process [11].

Various methods have been explored for bug severity prediction in the literature to enhance software maintenance efficiency and product quality. One way of assigning severity to a bug is through automated classification of the text description of a bug report. This approach utilizes text mining techniques to predict bug severity by analyzing both content and sentiment aspects of bug reports, resulting in improved accuracy and performance [12]. Several studies have proposed solutions for automated bug severity prediction and are mostly focused on machine learning approaches [5], [8], deep learning methods [11], [13], [6] and the use of word embeddings [14]. Recently, a few studies have employed lexicons or word lists [6], [15], [16] either as sources of features for machine learning or as baseline approaches for comparison. However, in almost all the cases these are not lexicons that would be tailor-made for the problem of bug severity classification, but lexicons that were designed for sentiment analysis. Classification approaches based on lexicons are usually very resource-friendly and can be employed in situations when the labeled data is scarce. They are usually outperformed by machine learning approaches but remain commonly used in some domains, for example in finance [17], and serve as baseline approaches to compare against.

In the field of bug severity classification, one of the rare works that provides insight into the performance of such approaches is the work of Barrah et al. [6], but the lexicon used there is a general sentiment one, like in many similar studies on bug severity. There seems to be a simple reason for this, namely, according to our literature and research review, there is no lexicon publicly available yet for bug severity classification. An attempt in this direction was the SentiStrength-SE lexicon [18], which is an adaptation of a general sentiment analysis lexicon to software development although focusing only on sentiment. The only specific lexicon of this kind that we could find was reported to be developed for an experimental comparison [19]. However, the lexicon that was used is not provided, and there are no insights about its standalone performance, as it was used as a feature filter for machine learning approaches, and the study

leaves a lot to be desired in terms of methodology and experimental setup.

The most recent systematic review of the literature conducted by Olaleye et al. [20] on predictive analytics and software defect severity highlights the growing interest in machine learning-based approaches for severity prediction, it identifies key areas that require further exploration, emphasizing the need for comprehensive models that can effectively capture the nuances of software defects. This aligns with ongoing efforts to refine classification techniques and improve the accuracy of severity assessments. Another study on software bug severity classification investigated the combination of source code metrics and large language models (LLM) [21]. The study proposed the integration of source code metrics into CodeBERT, to enhance its performance in classifying bug severity. The dataset used consisted of bug reports from various software projects, which were analyzed using machine learning models such as Decision Tree and Random Forest. The results indicated that fine-tuning CodeBERT LLM significantly enhanced bug severity prediction results compared to traditional models, and integrating source code metrics into CodeBERT further improved the model's performance in bug severity prediction. Moreover, Guo et al [22] further advanced the field by addressing the issue of distribution imbalance in bug report datasets. Their approach combines CR-SMOTE (a synthetic minority over-sampling technique) with an Extreme Learning Machine (ELM), indicating robust performance against real data imbalance, which is a common challenge in software bug classification. Their findings emphasize the necessity of addressing data imbalance to improve classification outcomes.

Chapter 3

Data

In this study, data has been sourced from software bug reports, which are documented in the bug tracking system and collected in CSV format. This chapter provides an overview of the data utilized in our research. We begin with an explanation of the datasets used and highlight their sources and characteristics. Following this, we explain the data preparation processes that were undertaken to ensure the appropriate characteristics of the data for our study.

3.1 Datasets

The dataset used in this study was taken from the online platforms Bugzilla[1] and Eclipse[2]. There are various columns in these datasets, but we have considered the default columns for our study, the column names are shown in Table 3.1.

Table 3.1: Column names in Eclipse and Firefox datasets

Eclipse		**Firefox**	
Bug ID	Product	Bug ID	Product
Component	Assignee	Component	Assignee
Status	Resolution	Status	Resolution
Summary	Changed-On	Summary	Updated
Priority	Severity	Priority	Severity
		Type	-

The column names in the datasets are mostly consistent, with a few exceptions; for example, 'Updated' in the Firefox dataset corresponds to 'Changed-on' in the Eclipse dataset. Additionally, the Eclipse dataset does not have a 'Type' column, as these values are integrated into the 'Severity' column. For instance, the 'Severity' column in the Eclipse dataset contains all the severity levels of a defect. However, if the data item is not a valid defect, the values are set to 'Enhancement' or 'Task' instead of a severity level. To ensure uniformity in column naming, we renamed 'Updated' in the Firefox dataset to 'Changed-On' and introduced a new 'Type' column in the Eclipse dataset. This 'Type' column has values of 'defect' if the 'Severity' column contains severity levels, and it is set to 'Enhancement' when the 'Severity' column in the data item is either 'Task' or 'Enhancement.' In this study, our primary focus is on the 'Severity' column, while the 'Summary' column serves as a source of features containing an unstructured textual description that provides context for the bug.

[1] `https://bugzilla.mozilla.org/`
[2] `https://bugs.eclipse.org/`

The severity levels in the target column are generally set as a blocker, critical, major, normal, minor and trivial, or in some cases, the severity level is assigned as S1, S2, S3, and S4 as explained in Table 3.4. The Type column contains information on topics such as defects, enhancement, or tasks. A defect here refers to a fault in the system whereas, enhancement represents a new feature in the system. Task on the contrary defines any refactoring, removing, replacement, enabling, or disabling of functionality. A general error report holds also various other information such as Bug ID, Status, Submission Date, etc., but the most critical attribute of the bug report is the severity level of a bug, on the basis of which one decides how rapidly it should be resolved. A bug report includes a detailed description of the observed issue, clear instructions on how to reproduce the bug, expected and actual behavior, and additional information such as screenshots, attachments, and code snippets. Bug reports also specify which software components are affected, such as bookmarks, history, or the rendering engine, which helps developers narrow down the issue and allocate resources effectively. A bug report undergoes various statuses throughout its lifecycle, including New, Verified, Resolved, and Won't Fix. The total number of data items in both datasets are shown in Table 3.2

Table 3.2: Summary of data items in the Eclipse and Firefox datasets. The Total column represents the total number of bugs collected. The Cleaned column shows the total number of data items after cleaning. The Severe and Non-Severe columns indicate the number of Severe and Non-Severe bugs, along with their percentages.

Dataset	**Total**	**Cleaned**	**Severe**	**Non-Severe**
Eclipse	31,115	27,393	6,555 (23.93%)	20,838 (76.07%)
Firefox	20,000	14,745	1,659 (11.25%)	13,086 (88.75%)

Figure 3.1 illustrates the distribution of bugs within these datasets, revealing a consistent trend: Severe bugs are less frequent compared to Non-Severe bugs. This observation aligns with the typical nature of bug reports, where Severe issues are prioritized for resolution, organizations strive to address Severe bugs promptly to ensure software reliability and enhance user experience.

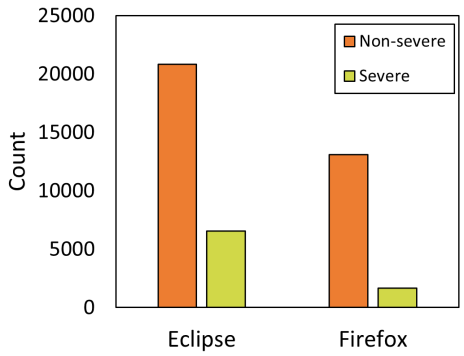

Figure 3.1: Distribution of bugs in Eclipse and Firefox dataset

The following table summarizes the details of Bug ID 123456 from the Firefox bug reports in Bugzilla, as shown in Table 3.3.

The severity levels in the bug report are shown in Table 3.4

Table 3.3: Sample data item from Firefox bug reports in Bugzilla

Field	Details
Bug ID	123456
Summary	Firefox crashes when opening multiple tabs. When opening more than 20 tabs simultaneously, Firefox crashes and closes unexpectedly. This issue has been observed multiple times and is reproducible on different machines.
Product	Firefox
Component	General
Version	89.0
Operating System	Windows 10
Severity	Major

Table 3.4: Bug Severity Labels, Levels, and Descriptions

Severity Label	Severity Level	Description
Blocker	Catastrophic	Blocks development/testing, impacts >25% of users, causes data loss, requires immediate resolution (no workaround).
Critical	Serious	Major functionality severely impaired, high impact, no satisfactory workaround.
Major	Serious	Major functionality impaired, high impact, but partial workaround may exist.
Normal	Normal	Blocks non-critical functionality, workaround exists.
Minor	Minor/Trivial	Cosmetic issues, low user impact, does not affect core functionality.
Trivial	Minor/Trivial	Very minor, typographical or UI issues, negligible user impact.

3.2 Data Preparation

Following the extraction of the bug reports, the first step of our data preprocesing involved data cleaning to ensure the integrity of the datasets. Data items categorized as ‘enhancements’ and ‘tasks’ were excluded, as they do not qualify as defects. Additionally, entries with severity values of ‘N/A’ or ‘-’ were removed, as these did not provide useful information for our analysis. This data cleaning process is essential, as it enhances the quality of the dataset and ensures that only relevant information is considered for further analysis, that facilitates accurate assessments of bug severity classification and improve the overall reliability of our experimental results. After excluding irrelevant data items, the combined dataset comprised 33924 Non-Severe bugs and 8214 Severe bugs.

Furthermore, to convert this into a binary classification problem, the severity levels in the ‘Severity’ column were categorized into two Classes: Severe and Non-Severe. Specifically, the severity levels of Blocker, Critical, and Major were labled as Severe, while the levels of Normal, Minor, and Trivial were labled as Non-Severe.

In the next step of the data preparation, the textual data in the summary column was preprocessed to transform it into structured data more suitable for analysis and modeling. These preprocessing steps involve tokenization, stopwords removal, and lemmatization.

Tokenization

Tokenization is the process of breaking down the text into individual tokens or words. In our study, tokenization is performed using Python module regular expressions (re) and the ‘split()‘ method after converting the text to lowercase and removing non-alphabetic characters. With such tokenization, we keep the process transparent and under our control without relying on

external libraries.

```
Example:
Input text:
    ‘Firefox crashes when opening a new tab after the update.’

1. After removing non-alphabetic characters:
    ‘Firefox crashes when opening a new tab after the update’

2. Convert to lowercase:
    ‘firefox crashes when opening a new tab after the updates’

3. Tokenize by splitting the text using split() method
   [‘firefox’, ‘crashes’, ‘when’, ‘opening’, ‘a’, ‘new’, ‘tab’, ‘after’, ‘the’,
   ’update’]
```

Stopwords Removal

Stopwords are common words, such as ‘the’, ‘a’, ‘an’, and ‘in’, that do not carry significant meaning and are often removed from the text to focus on more informative words. In our study, stopwords were identified and removed using the ‘stopwords‘ list from the ‘nltk.corpus‘ module. The word ‘not’ is treated specially to preserve its context by concatenating it with the following word while handling negation. By removing these stopwords, we were able to focus on the more informative and relevant terms within the bug report summaries.

```
Example continued:
Remove stopwords:
stopwords: [...‘when’, ‘a’, ‘new’, ‘after’, ‘the’...]
Remaining Tokens: [‘firefox’, ‘crashes’, ‘opening’, ‘tab’, ‘update’]
```

Lemmatization

Lastly, we applied the lemmatization technique to the tokenized and stopwords-removed text. It is a normalization process that converts words to their base or root form, known as a lemma. This step helps to account for word variations and ensures that semantically similar terms are treated equally, improving the overall quality of the textual features. For example, the words ‘running’, ‘ran’, and ‘run’ would all be converted to their lemma, ‘run’, through the lemmatization process. In our case the ‘WordNetLemmatizer‘ from the ‘nltk.stem‘ module is used to perform lemmatization.

```
Example continued:
Lemmatized Tokens: [‘firefox’, ‘crash’, ‘open’, ‘tab’, ‘update’]
```

These preprocessing steps were crucial in our study, as they enabled us to extract meaningful features from the textual data. By applying these techniques, including tokenization, stopwords removal, and lemmatization, the unstructured bug report summaries were transformed into a structured data format in which the feature values are the frequencies of the terms (tokens). Such a representation is then used in our further analysis and modeling in various experiments.

Frequent words appearance

The table below displays the top 25 most frequent words in the summary column of the both Eclipse and Firefox dataset.

Top 25 frequently appearing words in Eclipse dataset

not: 4664, project: 3321, file: 2320, web: 2208, xml: 2045, server: 1835, editor: 1828, eclipse: 1701, validation: 1694, error: 1504, add: 1270, java: 1255, jsp: 1233, org: 1196, need: 1114, npe: 1141, j: 1071, create: 1024, ee: 1002, service: 929, fail: 836, wtp: 810, ear: 831, import: 709...

Top 25 frequently appearing words in Firefox dataset

calendar: 2936, not: 2728, event: 1430, browser: 1148, lightning: 1047, tab: 942, events: 910, firefox: 885, new: 884, open: 842, view: 786, show: 773, test: 752, time: 735, task: 734, file: 732, js: 718, display: 683, use: 692, button: 661, window: 665, page: 647, work: 620, error: 605, change: 564, menu: 559, ...

Chapter 4

Methodology

This chapter outlines the research methodology, beginning with lexicon-based methods, which include different approaches to creating lexicons and the lexicon-based classification approach, followed by other classification methods. We included classical machine learning methods in our evaluation to assess the effectiveness of our proposed approach. Finally, the empirical settings of this study are presented, focusing on the performance metrics employed to evaluate the models and methods.

Note that in this thesis, we use the term *class* to refer to the target labels in the dataset (e.g., Severe and Non-Severe), *group* when discussing collections of bugs according to their severity in the context of the bug reports, and *category* to describe the Severe and Non-Severe divisions in the final lexicons.

4.1 Lexicon-Based Methods

Lexicon-based approaches use predefined lists of words or phrases to analyze text. Lexicons can be developed through various methods, such as identifying frequently occurring words in a specific domain to create domain-specific lexicons. Another method involves constructing lexicons based on semantic fields, and grouping words by meaning or context. Many lexicons are specifically designed for sentiment analysis, identifying words that convey positive, negative, or neutral sentiments. The advantages of lexicon-based methods include their simplicity, interpretability, and efficiency, making them suitable for rapid and scalable text analysis. Additionally, training data is not a necessary requirement for lexicon-based methods, which is beneficial in scenarios with limited or no labeled data.

4.1.1 Lexicon Creation

There are various approaches to lexicon creation. In the following, we introduce two common ones and our own threshold-based approach.

4.1.1.1 Manual Approach

Manual selection of terms that are indicative of the categories of interest (in this case the *Severe* and *Non-Severe*) is a lexicon creation approach that ensures complete human control of the process but is at the same time labor intensive [23]. Manually created lexicons usually contain a relatively low amount of terms, which tend to be mostly free of noise and errors. Such small and high-quality lexicons are commonly used as seed resources in various approaches, such as lexicon enrichment or feature construction for machine learning. A simple example of lexicon enrichment is the extension of the sets of terms with their synonyms and hypernyms using language resources such as WordNet [24]. More elaborate approaches use vector embeddings [25]

or language models [26]. Such extension procedures and their quality checking can be automated to various degrees.

4.1.1.2 Linear SVM-Based Approach

A Linear SVM (Support Vector Machine) is a supervised machine learning algorithm that finds the optimal linear decision boundary (in general, a hyperplane) to separate data points of different classes. It works by maximizing the margin—the distance between the hyperplane and the closest data items from each class, which are denoted as support vectors. This ensures the best possible separation of classes according to the maximum margin principle.

In addition, a linear SVM can be leveraged to identify important features, such as the indicative words of each class. This way machine-learned SVM data models can also be exploited for lexicon creation. For use in our work, we created lexicons with linear SVM-based approach, following the approach from a paper by Islam et al. [27]. The summary text column from the training dataset was pre-processed with NLP techniques to obtain a corpus of words. The CountVectorizer method was used to transform the processed text into a numerical word matrix. Through training and testing with different values of parameter c on a validation set, the best parameter value was determined in terms of the F1-score. The linear SVM model with the best performing c parameter was used to gather the information on the coefficients of features, with positive coefficients used as indicative of the target class (Severe), and negative coefficients for the opposite class (Non-Severe). Consequently, two distinct word lists were established: for the Severe category comprising words with positive coefficients, and for the Non-Severe category, including words with negative coefficients.

The Severe category is then created based on Equation 4.1 and the Non-Severe category is created based on Equation 4.2.

$$Coef_{\mathrm{s}} > 0, \tag{4.1}$$

$$Coef_{\mathrm{ns}} < 0, \tag{4.2}$$

where $Coef_{\mathrm{s}}$ denotes the coefficients for Severe words, and $Coef_{\mathrm{ns}}$ denotes the coefficients of Non-Severe words.

4.1.1.3 Threshold-Based Approach

In this work, a new approach to the creation of lexicons was designed for experimentation. It is shown in Figure 4.1 and works as follows. The training dataset is preprocessed through tokenization, stopwords removal, and lemmatization using a custom function. These transformations are applied exclusively to the Summary column. The dataset is then split into two subsets (Severe and Non-Severe) based on their severity labels. For each subset, the preprocessed summaries are converted into lists of individual words, and the frequency of each word is calculated using Python's Counter function. This generates two wordlists, one for each severity category, detailing how often each word appears. Next, the two lists are merged into a single set to get a list of unique words. For each word w in this combined set, it's frequency in both the Severe and Non-Severe categories is denoted by $C_{\mathrm{s}}(w)$ and $C_{\mathrm{ns}}(w)$, respectively. If a word is absent in either category, its count is assigned a value of 0. Finally, two ratios are computed for each unique word based on these counts.

The ratio of appearance of the word w in the Severe category is computed as shown in Equation 4.3.

$$R_{\mathrm{s}}(w) = \frac{C_{\mathrm{s}}(w)}{C_{\mathrm{s}}(w) + C_{\mathrm{ns}}(w)}, \tag{4.3}$$

Similarly, the ratio of appearance of the word w in the Non-Severe category is calculated as defined in Equation 4.4.

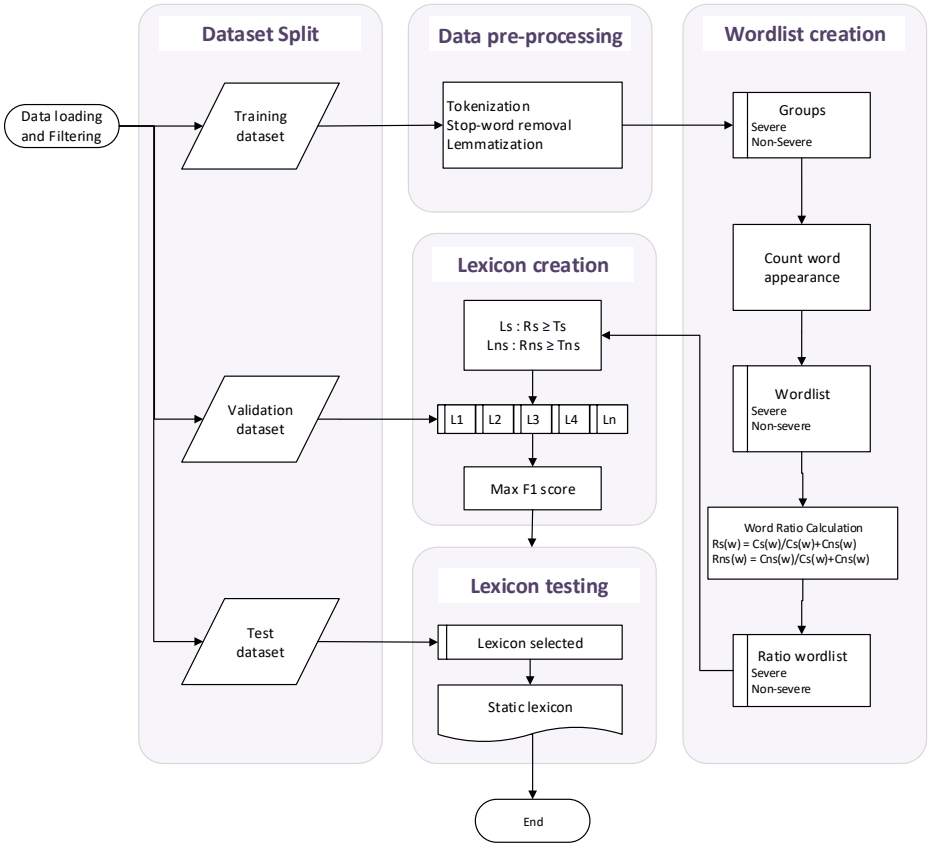

Figure 4.1: Workflow for threshold-based lexicon creation: This diagram outlines the process for constructing and validating a lexicon.

$$R_{\mathrm{ns}}(w) = \frac{C_{\mathrm{ns}}(w)}{C_{\mathrm{s}}(w) + C_{\mathrm{ns}}(w)}, \tag{4.4}$$

The lexicons L_{s} and L_{ns}, for Severe and Non-Severe bugs, respectively, are then defined based on a given threshold, for Severe in line with Equation 4.5 and for Non-Severe in line with Equation 4.6

$$L_{\mathrm{s}} = \{w \in W; R_{\mathrm{s}}(w) \geq T_{\mathrm{s}}\}, \tag{4.5}$$

$$L_{\mathrm{ns}} = \{w \in W; R_{\mathrm{ns}}(w) \geq T_{\mathrm{ns}}\}, \tag{4.6}$$

where $R_s(w)$ and $R_{ns}(w)$ are the severity and non-severity ratios of each word w from the word list W, while T_s and T_{ns} denote the thresholds, which are parameters of our method. Their values, ranging between 0.1 and 1.0 in our experiments, are optimized on a validation subset of the learning data using the F1-score. The thresholds that maximize performance are selected to generate the final lexicons. This method produces two distinct lexicons: Severe and Non-Severe. These lexicons are rigorously evaluated on the test dataset. Finally, both lexicons are combined into a single JSON file, preserving their respective categories.

4.1.2 Lexicon-Based Classifiers

Given a lexicon, there are many ways it can be used for classification. This section explains the lexicon-based classification approach that we used and provides an example that demonstrates

how the classifier works in various classification scenarios. In addition, we discuss some other lexicon-based classification methods.

4.1.2.1 Our Lexicon-Based Classification Approach

The classification approach for the lexicon-based classifier that we used was very basic and straightforward, as shown in Figure 4.2.

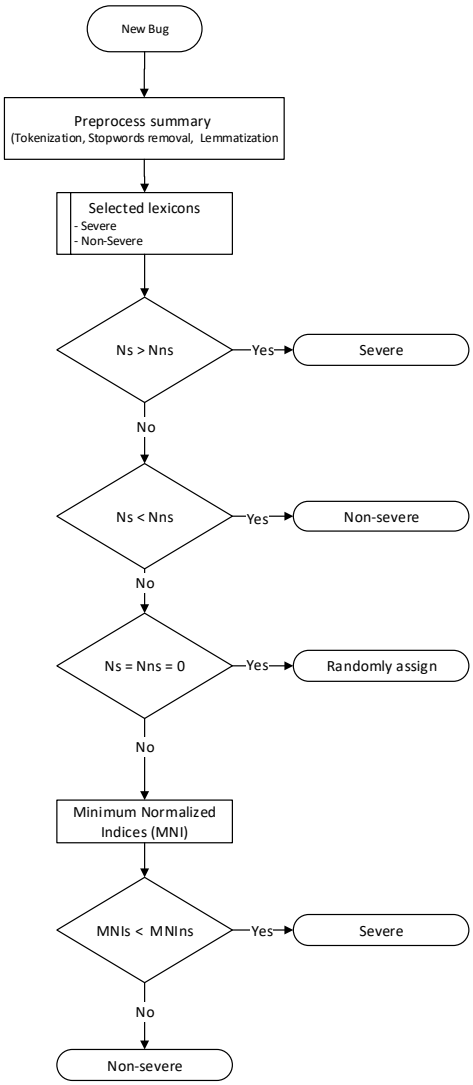

Figure 4.2: Our lexicon-based classifier: Each scenario leads to an outcome, which is represented by the rounded rectangles.

A new bug was classified into the category that corresponded to the lexicon with which the intersection of the words was the largest.

$$N(B, L) = |\{w \in B \cap L\}|, \tag{4.7}$$

where the function $N(B, L)$ calculates the number of words w that are common between the bug report B and the lexicons L, and for a given bug report B, $N_{\text{s}} = N(B, L_{\text{s}})$ and $N_{\text{ns}} = N(B, L_{\text{ns}})$.

When a bug summary has an equal, non-zero number of intersecting words with both the Severe and Non-Severe lexicons, it is classified as a neutral bug. To further classify these neutral bugs, we first sort the lexicons in descending order of their severity/non-severity ratios. For each intersecting word, we calculate its normalized index in both sorted lexicons shown in Equation 4.8. This index reflects the word's relative ranking within each lexicon, where a lower normalized value (due to a smaller index) corresponds to a higher-ranked word (more Severe or Non-Severe). These normalized indices are stored in two separate lists for Severe and Non-Severe normalized indices.

$$NI(w, L) = \frac{I(w, L)}{|L|} 100, \tag{4.8}$$

where I(w, L) is the index (rank) of the word w in lexicon L and |L| is the size of the lexicon L. Next, we converted these lists into two data frames: one for Severe normalized indices and another for Non-Severe normalized indices. For each data frame, we calculated the minimum normalized index value. This allowed us to identify the category (Severe or Non-Severe) with the lowest minimum value, which we hypothesized would best reflect the bug report's true severity level. Finally, we compared the minimum values of both categories, assigning the bug report to the category with the lower minimum index as its predicted severity.

$$MNI(B, L) = \min \{NI(w, L); w \in B\}, \tag{4.9}$$

where each word w in B that appears in L, its normalized index $NI(w, L)$ is computed based on its position in L, and $MNI(B, L)$ is defined as the minimum of these values and $MNI_{\text{s}} = MNI(B, L_{\text{s}})$ and $MNI_{\text{ns}} = MNI(B, L_{\text{ns}})$. In cases of neutral bugs where the intersection of words between the bug summary and the Severe and Non-Severe lexicons is equal and the intersection count is zero, we employ a random assignment approach to determine the severity level. Specifically, the bug report is randomly assigned to either the Severe or the Non-Severe category.

It is important to note that this random assignment approach was utilized as a last resort when all other lexicon-based classification methods had been exhausted. Moreover, this random assignment was only applied to a small subset of neutral bugs. By incorporating this strategy for zero-equal ratio neutral bugs, we aimed to provide a comprehensive and robust lexicon-based approach for bug severity classification. Ultimately, this method ensured that all bug reports were assigned a severity level, even in cases where the textual content alone did not provide a clear indication of the bug's criticality.

Example of Our Lexicon-Based Classifier

Our lexicon-based classifier determines bug severity by analyzing the intersection of words between a bug report and the Severe/Non-Severe lexicons. Table 4.1 illustrates four classification scenarios, demonstrating how the method handles cases where word counts differ or overlap. For each scenario, the table displays the original bug summary (e.g., 'payment gateway integration fails blocking user transaction'), its preprocessed tokenized form (Summary list), the intersecting Severe/Non-Severe words, and their respective counts (N_s and N_{ns})). In cases where counts are equal and non-zero (e.g., Scenario 3), the table includes the minimum normalized indices (MNI_s and MNI_{ns}), which reflect word rankings in the sorted lexicons. The predicted severity

is assigned by comparing word counts or, for tied scenarios, the minimum normalized indices, prioritizing the category with the lower index to resolve ambiguity.

Table 4.1: Lexicon-based classifier example. This table demonstrates the various scenarios handled by our developed classifier method for classifying new software bugs as severe or non-severe.

Classification: Scenario 1
Bug: *payment gateway integration fails blocking user transaction*
Summary list: {*'payment', 'gateway', 'integration', 'fail', 'block', 'user', 'transaction'*}
Severe words: {*'user', 'fail', 'transaction'*}
Non-severe words: {}
Severe word count (N_s): 3
Non-severe word count (N_{ns}): 0
Predicted Severity: **Severe**
Classification: Scenario 2
Bug: *spelling mistake text footer section*
Summary list: {*'spell', 'mistake', 'text', 'footer', 'section'*}
Severe words: {}
Non-severe words: {*'mistake', 'section'*}
Severe word count (N_s): 0
Non-severe word count (N_{ns}): 2
Predicted Severity: **Non-severe**
Classification: Scenario 3
Bug: *security vulnerability allowing unauthorized access web application*
Summary list: {*'security', 'vulnerability', 'allow', 'unauthorized', 'access', 'web', 'application'*}
Severe words: {*'unauthorized'*}
Non-severe words: {*'unauthorized'*}
Severe word count (N_s): 1
Non-severe word count (N_{ns}): 1
Minimum severe normalized index (MNI_s): 21.875
Minimum non-severe normalized index (MNI_{ns}): 68.75
Predicted Severity: **Severe**
Classification: Scenario 4
Bug: *Firefox slowdowns when opening multiple tabs*
Summary list: {*'firefox', 'slowdowns', 'open', 'multiple', 'tab'*}
Severe words: {}
Non-severe words: {}
Severe word count (N_s): 0
Non-severe word count (N_{ns}): 0
Predicted Severity: **Non-severe**

4.1.2.2 Other Lexicon-Based Classification Methods

Lexicon classification can also be done in other ways. Particularly interesting are some intensity-based methods that are common in the domain of sentiment analysis. Sentiment lexicon classification in such cases relies on a set of phrases that have specific sentiment orientations (SO) and their strengths to assess the overall sentiment of a document. This approach assigns positive SO values to favorable expressions and negative values to unfavorable ones. For instance, the word

'good' could carry an SO value of +1, whereas 'not good' would be rated at -1. Techniques also incorporate intensifiers and diminishers, like 'very good' (which is more positive) and 'barely good' (which is less positive), thus enabling more nuanced SO values that range from -5 to +5 [28] [29].

Another example is VADER (Valence Aware Dictionary and Sentiment Reasoner). It employs a pre-established lexicon containing over 7500 words and phrases. Each paired sentiment intensity score ranges from -4(highly negative) to +4(highly positive). VADER classifiers analyze text by breaking it into individual words, assigning sentiment scores to each, and then calculating the overall score. Its rule-based features enhance sentiment analysis by effectively handling negation and incorporating grammatical conventions. For instance, it reverses sentiment scores for phrases that are negated, such as 'not good', and modifies scores according to intensity boosters, like 'very' or 'barely.' Furthermore, VADER takes punctuation and capitalization into account to enhance the accuracy of sentiment assessments and in fact, it is not just a lexicon-based method [30].

4.2 Machine-Learning Methods

Though several studies on software bug severity classification have used machine learning models, we focused on Support Vector Machine (SVM), Logistic Regression (LogReg), and Naïve Bayes models (multiNB) in this work due to their distinct characteristics. These models were evaluated for their effectiveness in predicting bug severity levels based on lexicon-derived features.

4.2.1 Support Vector Machines

Support Vector Machines are widely used supervised learning algorithms known for their effectiveness in handling high-dimensional data spaces. SVMs are particularly suited for binary classification tasks as they are binary classifiers in essence, identifying the maximum margin hyperplane that separates two classes in the feature space. SVMs have the ability to handle both linearly separable and non-linearly separable data using kernel functions, which transform the original feature space into a higher-dimensional space where the data becomes linearly separable. The SVM approaches are also very efficient in terms of handling large feature sets, making them a suitable choice for tasks that involve a large number of features, such as text classification, which is the task at hand in our study. The combination of strong theoretical foundations, versatility, and computational efficiency has made SVM a widely adopted algorithm in various domains including machine learning, pattern recognition, and data mining.

In our study, we used the linear SVM classifier to develop a predictive model with the same data split that was utilized for dynamic lexicon creation. We used the SVM classifier implementation from the scikit-learn library with default parameters ($c = 1$).

4.2.2 Logistic Regression

Logistic Regression, a widely used statistical model for binary classification tasks, like bug severity prediction, estimates the probability of a data item belonging to a specific class. Known for its simplicity, interpretability, and computational efficiency, logistic regression is also quite robust to noise. These qualities make it a sensible approach for our experiments with bug severity classification, particularly as it offers efficient model training and classification.

In our study, we used the scikit-learn implementation of logistic regression with default parameters on the same data split utilized for dynamically creating the lexicons.

4.2.3 Naïve Bayes

Naïve Bayes, a probabilistic classifier based on Bayes' theorem with the assumption of feature independence, is well known for its simplicity and efficiency in text classification tasks. This classifier excels in handling high-dimensional data and offers rapid training with minimal computational resources and seamlessly manages missing data. It stands out for its effectiveness in text classification, making it a valuable tool for various applications requiring efficient and reliable classification algorithms.

In our study, we used the Multinomial Naïve Bayes implementation from the scikit-learn library on the same data split as all other approaches.

4.3 Empirical Evaluation

For this study, we empirically evaluated the lexicon learning and classification approach and three classic machine learning methods: Support Vector Machine (SVM), Logistic Regression, and Naïve Bayes in eight experimental scenarios. In each experiment, the dataset was randomly split into training, validation, and testing datasets. The lexicon-based classifier was evaluated using tests with different Severe and Non-Severe thresholds. The same dataset-split was then used to train the machine learning (ML) classifiers. Each experiment was repeated ten times, with a diverse random split each time.

4.3.1 Performance Assessment Metrics

The F1-score is a critical metric for evaluating classification models, particularly in imbalanced datasets. It balances precision (accuracy of positive predictions) and recall (completeness of positive predictions), making it ideal for scenarios where both false positives (FP) and false negatives (FN) must be minimized. These metrics are especially relevant to our work, as our dataset exhibits class imbalance between Severe and Non-Severe bug reports. Below, we define the confusion matrix components and metrics used:

- **True Positive (TP)**: Actual class is *Severe*, predicted class is *Severe*.
- **False Positive (FP)**: Actual class is *Non-Severe*, predicted class is *Severe*.
- **True Negative (TN)**: Actual class is *Non-Severe*, predicted class is *Non-Severe*.
- **False Negative (FN)**: Actual class is *Severe*, predicted class is *Non-Severe*.

The metrics are formally defined as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}, \tag{4.10}$$

where accuracy measures the proportion of correctly classified instances.

$$Precision = \frac{TP}{TP + FP}, \tag{4.11}$$

where precision quantifies the fraction of true positives among all predicted positives.

$$Recall = \frac{TP}{TP + FN}, \tag{4.12}$$

where recall measures the fraction of true positives identified from all actual positives.

$$F1\text{-}score = 2 \times \frac{Precision \times Recall}{Precision + Recall}, \tag{4.13}$$

where the F1-score is the mean of precision and recall.

$$F1\text{-}mean = \frac{F1\text{Severe} + F1\text{Non-Severe}}{2}, \tag{4.14}$$

where the F1-mean averages the F1-scores of both classes, ensuring equal weight for Severe and Non-Severe categories.

Chapter 5

Experimental Setting

This chapter presents a series of experiments conducted to generate a lexicon for bug severity classification. The first four experiments were conducted using varying combinations of severe and Non-Severe thresholds to dynamically generate lexicons for bug severity classification. Based on the performance outcomes of these experiments, it was determined that it is sensible to develop a static lexicon for this domain with different approaches, as the results were comparable with the classic machine learning models used in our experimentation. Furthermore, a group of experiments was conducted to develop static lexicons for this domain. The first four experiments were conducted to develop lexicons with a threshold-based approach and various combinations of datasets, another two experiments were conducted to develop a static lexicon with linear-SVM coefficient-based approach. Finally, four additional experiments were performed using various publicly available lexicons to establish a baseline solution for comparison. As shown in Table 5.1, a series of experiments was conducted employing various approaches and dataset combinations.

5.1 Dynamic Lexicons

In the first stage of our study, a group of experiments was conducted to create and assess dynamic lexicons, which means lexicons generated and adjusted in real-time based on varying thresholds during the experimentation process. By employing various combinations of thresholds, the lexicon is tailored to reflect the context of the dataset.

In Experiment 1, the lexicon was created and tested exclusively on the Eclipse dataset, followed by the training and testing of machine learning classifiers (SVM, Logreg, and MultiNB) on the same dataset split. This approach aimed to evaluate the performance of the lexicon method specifically on the Eclipse dataset. A unique characteristic of the Eclipse dataset is that its bug reports are written by developers and are technical in nature. Additionally, numerous studies on bug severity have utilized the Eclipse dataset, enabling us to compare the effectiveness of our classifier with previously reported findings. In Experiment 2, the lexicon was created and tested on the Firefox dataset alone, followed by training and testing of the ML classifier on the same split of the dataset. This experiment was conducted to observe the performance of the lexicon approach on Firefox dataset. In Experiment 3, the lexicon was created and tested on the combined dataset, followed by training and testing of the ML classifiers on the same split of the combined dataset. This was conducted to assess the performance of the lexicon approach on datasets from diverse sources. For this purpose, both datasets were merged in the pre-processing step so that both the learning and the prediction datasets have instances from both of the datasets. In Experiment 4, the combined datasets of Eclipse and Firefox were utilized, with bugs classified as 'Normal' excluded from the dataset. This exclusion minimized the imbalance in the distribution of data between the Severe and Non-Severe categories. Such exclusion is very common in related work.

For Experiments 1, 2, 3, and 4, the data was split into approximately 64% training, 16%

validation, and 20% testing sets across 10 iterations. This was achieved using scikit-learn's train_test_split in a two-step process: first, 20% was set aside for testing. The remaining 80% was then further split, yielding the 64% training and 16% validation portions. Each iteration employed a unique random seed for varied splits.

5.2 Static Lexicons

A group of cross-dataset experiments was conducted to create and test a general static lexicon for software bug severity classification, which means a lexicon that remains fixed.

Threshold-based lexicon

Experiment 5 was to assess the efficiency of the lexicon developed in the Eclipse dataset and verify its performance on a different dataset, namely Firefox. In this study, the Eclipse dataset was divided into 80% for training and 20% for validation, whereas, the entire Firefox dataset was utilized for testing purposes. After testing each generated lexicon with the validation dataset, the most suitable lexicon was selected in terms of F1-score (severe), and a static lexicon was subsequently generated from the lexicon with the highest F1-score. The output from this experiment is a static lexicon in JSON format and is called lexicon_THR_E. The objective of Experiment 6 was to evaluate the efficacy of the lexicon created on the Firefox dataset and to test it entirely on the Eclipse dataset. The Firefox dataset was split into 80% for training and 20% for validation, with 100% of the Eclipse data utilized for testing. The result of this experiment was a static lexicon in JSON format, referred to as lexicon_THR_F.

Linear SVM-based Lexicon:

In Experiment 9, the Eclipse dataset was split into 80% for training and 20% for validation to develop a lexicon using a Linear SVM coefficients approach. The developed lexicon was then tested on the Firefox dataset. In Experiment 10, the Firefox dataset was similarly divided into 80% for training and 20% for validation to create a lexicon and tested on the Eclipse dataset. The Linear SVM model was trained and tested on the validation dataset on different c parameters, the best hyperparameter c was identified based on the maximum F1-score, and the linear SVM model was fitted using this optimal c parameter value. Finally, the coefficients were obtained from the trained model, resulting in a list of words with positive and negative coefficients. The output from these experiments includes two static lexicons in JSON format: lexicon_SVM_E for the Eclipse dataset and lexicon_SVM_F for the Firefox dataset. In these lexicons, negative coefficients correspond to Non-Severe words, while positive coefficients indicate severe words. Since the target class is 'Severe', stronger positive coefficients signify a greater likelihood of belonging to this class according to the Linear SVM model. Overall, the data split for these experiments consisted of 80% for training, 20% for validation from one dataset, and the testing dataset on an entirely different dataset.

5.3 Publicly Available Lexicons

Following experiments were conducted to classify bug reports using an existing lexicon, establishing a baseline solution for comparison with our created static lexicon for the same task.

Bing Liu Lexicon

In Experiment 11, the complete Firefox dataset was utilized as the test dataset and in Experiment 12, the Eclipse dataset was used as a test dataset. Bing Liu's lexicon consists of two text files: one containing words with positive sentiments and the other containing words with negative

Table 5.1: Overview of the experiments.

	Training	Validation	Testing	Normal Status	Split
Dynamic Lexicons					
Experiment 1	Eclipse	Eclipse	Eclipse	Included	64|16|20
Experiment 2	Firefox	Firefox	Firefox	Included	64|16|20
Experiment 3	Eclipse&Firefox	Eclipse&Firefox	Eclipse&Firefox	Included	64|16|20
Experiment 4	Eclipse&Firefox	Eclipse&Firefox	Eclipse&Firefox	Excluded	64|16|20
Static Lexicons					
Experiment 5	Eclipse	Eclipse	Firefox	Included	80|20|100
Experiment 6	Firefox	Firefox	Eclipse	Included	80|20|100
Experiment 7	Eclipse&Firefox	Eclipse&Firefox	Eclipse&Firefox	Included	64|16|20
Experiment 8	Eclipse&Firefox	Eclipse&Firefox	Eclipse&Firefox	Excluded	64|16|20
Experiment 9	Eclipse	Eclipse	Firefox	Included	80|20|100
Experiment 10	Firefox	Firefox	Eclipse	Included	80|20|100
Existing Lexicons					
Experiment 11	-	-	Firefox	Included	No Split
Experiment 12	-	-	Eclipse	Included	No Split
Experiment 13	-	-	Firefox	Included	No Split
Experiment 14	-	-	Eclipse	Included	No Split

sentiments [23] [28]. In this experiment instead of our custom-developed lexicons, the Bing Liu's opinion lexicon was employed, the positive file was used as the nonsevere dictionary, and the negative file was used as a severe dictionary, and thus the classifier method that we created explained in section 4.1.2.1 is applied to classify a new bug. The effectiveness of this approach was then evaluated using the F1-score.

VADER Rule Based Lexicon

In Experiment 13, the Firefox dataset was utilized for bug severity classification using the VADER rule-based lexicon. In Experiment 14, the Eclipse dataset was employed for the same classification task with the VADER lexicon [30]. The VADER lexicon library was imported from NLTK and used for classification instead of the custom lexicon. A complete sentence from the description of a bug report was passed to the VADER classifier method, which leverages a *built-in function called 'analyzer.polarity_scores'*. This method generates polarity scores for the text, categorizing it as positive, negative, neutral, or compound. The compound score was then used to classify the bug reports into Severe and Non-Severe categories. If the compound score is greater than zero, the bug is tagged as Severe; if it is less than zero, the bug is tagged as Non-Severe. Finally, the classifier's performance was evaluated using the scikit-learn library.

5.4 Static Final Lexicons

Four static lexicons were created for software bug severity classification from different approaches and various experimental settings. The first three lexicons are created from the threshold-based approach, the fourth lexicon is created from the linear SVM-based approach. This section outlines details about the final static lexicons.

Final Static Lexicon: SEV_THR_LEX

The static lexicons generated from Experiments 5 and 6 on a cross dataset (lexicon_THR_E and lexicon_THR_F) were merged in JSON format. The merging process involved combining the 'Severe' category from lexicon_THR_E with the 'Severe' category from lexicon_THR_F, and similarly for the 'Non-Severe' categories. After merging, duplicates were checked within

each category. In the merged final lexicon, words with higher ratios $R_s(w)$ and $R_{ns}(w)$ were retained while duplicates with lower ratios were removed from each category. The filtered lexicon, after duplicate removal, is named `SEV_THR_LEX`, which contains 1,231 severe words and 6,719 Non-Severe words. An example of 10 random words from this final lexicon is shown in Table 5.2.

Final Static Lexicon: SEV_THR_LEX_c

The static lexicon `SEV_THR_LEX_c` was derived from Experiment 7. In this experiment, the Eclipse and Firefox datasets were first concatenated into a single, combined dataset. This combined dataset was then split following the methodology detailed in Section 5.1 (which as now clarified results in approximately 64% training, 16% validation, and 20% test data for each of the 10 iterations). The train_test_split function, applied to this entire merged dataset, inherently shuffles the data before partitioning. This random shuffling ensures that instances from both original source projects (Eclipse and Firefox) are randomly distributed across the training, validation, and test sets in each iteration, reflecting their proportions in the overall combined data. This lexicon was then constructed exclusively using the 64% training portion of these consequently mixed splits. The final lexicon contains 2,130 severe words and 5,479 Non-Severe words. An example of 10 random words from this lexicon is shown in Table 5.3.

Final Static Lexicon: SEV_THR_LEX_f

The static lexicon `SEV_THR_LEX_f` was generated from Experiment 8. For this experiment, bug reports with the status 'Normal' were first excluded from the combined Eclipse and Firefox dataset, creating a filtered, merged dataset. This merged filtered data set was then subjected to the same 10-iteration splitting process detailed in Section 5.1 (producing approximately 64% training, 16% validation and 20% test data). Similarly, the train_test_split function's inherent shuffling of this entire filtered, merged dataset ensured a random distribution of instances from both source projects into the training, validation, and test sets. The lexicon was built solely from this 64% training portion of these mixed splits. The final lexicon contains 45 severe words and 764 Non-Severe words. An example of 10 random words from this lexicon is shown in Table 5.4.

Final Static Lexicon: SEV_SVM_LEX

The static lexicons generated from Experiments 9 and 10 (lexicon_SVM_E and lexicon_SVM_F) were merged in JSON format. The merging process involved combining the 'Severe' category of lexicon_SVM_E with the 'Severe' category of lexicon_SVM_F and likewise for the 'Non-Severe' categories. The final lexicon from this experiment is called SEV_SVM_LEX. It contains 3,756 severe words and 7,307 Non-Severe words. An example of random 10 words from this final lexicon is shown in Table 5.5.

Table 5.2: Static Lexicon: SEV_THR_LEX

Severe Terms		Non-Severe Terms	
Term	Ratio	Term	Ratio
npe	0.5	webapps	1.0
create	0.306	clean	1.0
fail	0.204	noteworthy	1.0
facet	0.311	produce	0.917
freeze	0.411	different	0.953
publish	0.562	byte	0.818
unable	0.4225	compiler	0.878
workspace	0.333	unused	1.0
tvt	1.0	variable	0.938
tct	0.636	allow	1.0
...			

Table 5.3: Static Lexicon: SEV_THR_LEX_c

Severe Terms		Non-Severe Terms	
Term	Ratio	Term	Ratio
undo	0.272	paste	0.851
fold	0.208	document	0.898
fix	0.221	wrong	0.862
deploy	0.576	behavior	0.88
project	0.346	tie	0.820
weblogic	0.444	quick	0.891
generic	0.246	validation	0.809
server	0.267	little	1.0
api	0.203	failure	0.802
provide	0.2	ear	0.842
...			

Table 5.4: Static Lexicon: SEV_THR_LEX_f

Severe Terms		Non-Severe Terms	
Term	Ratio	Term	Ratio
leave	1.0	prevent	1.0
moz	1.0	keyboard	1.0
pdf	1.0	focus	1.0
drain	1.0	item	1.0
individual	1.0	overflow	1.0
structure	1.0	menu	1.0
bookmarkfolder	1.0	dot	1.0
restore	1.0	line	1.0
poor	1.0	focus	1.0
quality	1.0	url	1.0
...			

Table 5.5: Static Lexicon: SEV_SVM_LEX

Severe Terms		Non-Severe Terms	
Term	Ratio	Term	Ratio
leave	6.920	ico	−6.51
damage	6.268	drag	−3.47
renstalling	6.150	outbox	−3.47
decouple	5.771	prtime	−3.47
team	5.709	testtodaypane	−3.47
smth	5.540	uitour	−3.47
webextensions	5.328	caliicalproperty	−6.94
backward	5.052	complete	−6.94
reload	4.844	cta	−6.94
mrangeend	4.714	got	−6.94
...			

Chapter 6

Results and Discussion

In this chapter, we first outline the results of our experimental assessment of dynamic lexicon generation. For these experiments, we present box plots to show the variation in performance across the train, validation, and test datasets. Next, we present the results of the experiments for static lexicons, where the focus is on evaluating variability in a cross-dataset setting using different training data subsets. Note that box plots are not shown for the static lexicon experiments, as the test dataset remains fixed in each iteration, and no fine-tuning is performed for the ML methods; for the publicly available lexicons, only testing is conducted. Then, we discuss the results of the experiments for static lexicons after handling negation. Finally, we present bar graphs for all experiments, to see the direct comparison of results across different approaches and before and after handling negation for all different approaches.

Before we focus on our results, we shall revisit the best results from the related work. Previous studies, such as the one by Bibyan et al. [31] on assessing the severity of software bugs using neural networks, achieved good results with the F1-score (Severe) for separated four components of the Eclipse dataset. In this paper they removed the bugs with the "Normal" bugs to reduce imbalance and ambiguity. They then grouped the remaining levels into two classes: "Severe" (Blocker, Critical, Major) and "Non-Severe" (Minor, Trivial). Their neural network achieved F1-scores of 71% for UI, 54% for Core, 71% for Debug, and 68% for Text.

Furthermore, Hamouri et al. [32] achieved the highest reported F1-scores of 94% on the Eclipse dataset and 93% on the Firefox dataset using Neural Networks. Their results show such an performance ordering: Neural Networks (94% F1-score on Eclipse), ensemble methods (87%-90%), SVM (82%), with Random Forest achieving 41%. Their methodology employed comprehensive NLP preprocessing, with TF-IDF feature extraction to capture semantic relationships, SMOTE balancing techniques to address class imbalance issues, and leveraged neural networks' capacity to learn complex non-linear patterns in the data. Their study utilized a 6-class severity classification scheme for bug severity assessment. However, the preprocessing steps were indicated to be done on data before the train-test split, which could be methodologically questionable.

6.1 Dynamic Lexicons Results

Results from our dynamic learned lexicon experimentation are presented in Tables 6.1–6.4. The best results of each performance measure are displayed in bold in this and all the following tables with performance results. Additionally, the results shown in the tables are presented in Figure 6.5, showing a graphical representation of the trends and insights in terms of F1-score and F1-mean. The tables present a comparative evaluation of various bug severity classification models, including a lexicon-based approach, Support Vector Machines, Multinomial Naïve Bayes, and Logistic Regression. The metrics reported include F1-score (Severe), F1-mean, True Positives (TP), False Positives (FP), True Negatives (TN), False Negatives (FN), computing time for

learning (tl), and computing time for classification (tc).

According to the F1-score for the Severe category as the most relevant performance measure in our setting—the usefulness of the lexicon approach is shown by its superior performance in three experiment variations and very comparable performance to ML approaches. However, there are at least two cautionary notes to consider in view of these results. First, as explained in Section 4.1.1.3, we employed a learning approach to learn and tune the lexicon to the training data, so it is not a classic lexicon-based classification approach, but machine-learning to some extent. Second, the SVM approach is very sensitive to some parameters (we used defaults), and a much more informative assessment of its performance can be gained if the parameters are tuned with the validation set, like in the case of the lexicon approach. The SVM tuning was left out because it took prohibitive amounts of time. However, we have the F1-score results of SVM with parameter fitting for Experiment 1 and it is 0.3314, considerably better than with defaults, but still comparable to other approaches.

The running time of the learning and classification parts of the approaches are also reported in the result, Tables 6.1–6.2 (in seconds). As mentioned, even without parameter tuning, the SVM took much longer than other methods. The learning part of our lexicon approach is also time-consuming, but as learning is not necessary given a fixed static lexicon, which is our end goal, the more important is the classification time. Interestingly, even during classification, some classic machine learning approaches are faster than our (which is not time optimized) use of the lexicon. This is a relevant result to consider, as besides usefulness in situations with low (or no) training data, lexicons are commonly argued to be fast and not computationally intensive.

Table 6.1: Results of Experiment 1 as averages of 10 iterations.

	F1-score	F1-mean	TP	FP	TN	FN	tl	tc
lexicon_THR	**0.4631**	0.5466	821.7	1392.4	2775.7	489.2	1024.6	5.2
SVM	0.2866	0.5737	256.1	220.1	3948	1054.8	2225.1	140.03
MultiNB	0.3559	**0.6041**	363.7	368.8	3799.3	947.2	1.2	0.04
LogReg	0.3030	0.5815	277.7	244.2	3923.9	1033.2	3.2	0.08

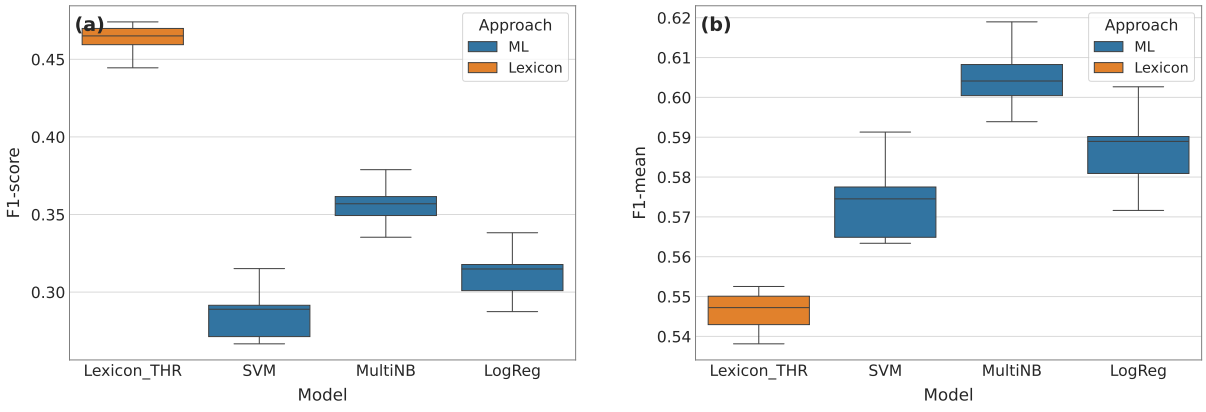

Figure 6.1: Box plots for Experiment 1 comparing the F1-score (a) and mean F1-score (b) between our threshold-based lexicon approach and machine learning methods.

The results of Experiment 1 are displayed in Table 6.1, and box plots are presented in Figure 6.1. The results show that the lexicon-based approach achieved the highest F1-score of 0.4631 and F1-mean of 0.5466, outperforming the machine learning models. This suggests

that the lexicon-based method is effective in capturing the textual patterns and characteristics associated with bug severity using the information contained in the Severe and Non-Severe word lexicons. On the other end, the machine learning methods, including SVM, MultiNB, and LogReg, were first trained on various sizes of feature sets: '1000, 10000, 15000, and unlimited', it was observed that after 15000 features there was no big difference in the performance metrics, the results show that the machine learning models exhibit F1-score 0.2866 to 0.3559 and F1-mean 0.5737 to 0.6041 when trained on the full, unlimited feature set. The results indicate that there are opportunities for enhancing the performance of machine learning models through feature engineering, and hyperparameter tuning. However, these models demonstrate higher F1-mean scores, ranging from 0.5737 to 0.6041. This means that the machine learning models are better at predicting the Non-Severe class, and the Non-Severe F1-score is better. However, we are interested in the F1-score for Severe.

The computation time analysis shows that SVM in most cases took the longest for both learning and classification. The shortest classification times were achieved by Multinomial Naïve Bayes (MultiNB) and Logistic Regression (LogReg). Our lexicon-based approach performed better than SVM in terms of both learning and classification time. However, it is slower than MultiNB and LogReg. Overall, our threshold-based approach provided a good balance between computational time and accuracy in this experiment setting.

Table 6.2: Results of Experiment 2 as averages of 10 iterations.

	F1-score	F1-mean	TP	FP	TN	FN	tl	tc
lexicon_THR	**0.3997**	0.5266	202.5	475.5	2140.9	130.1	333.3	1.7
SVM	0.2759	0.6061	65.9	78.6	2537.8	266.7	83.9	16.4
MultiNB	0.3382	**0.6345**	94.5	131.5	2484.9	238.1	0.5	0.02
LogReg	0.2985	0.6193	69.3	61.6	2554.8	263.3	1.3	0.03

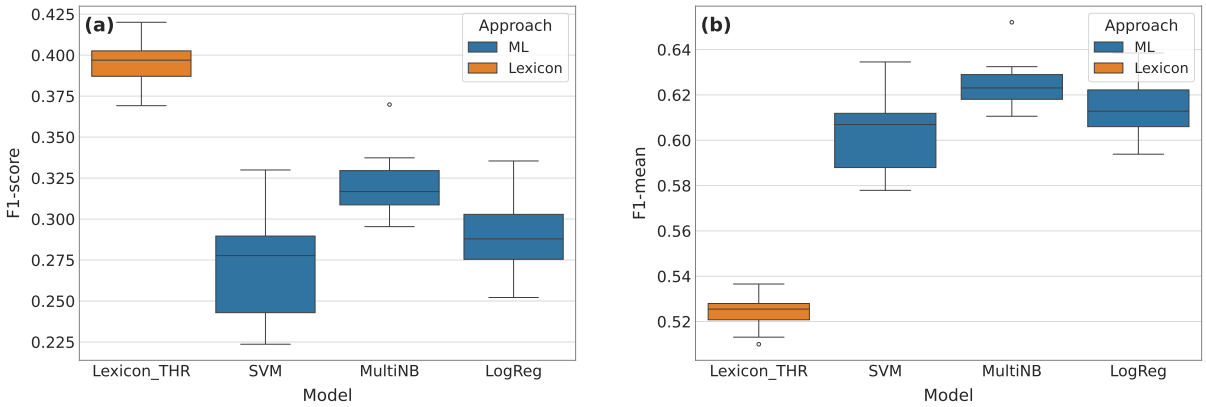

Figure 6.2: Box plots for Experiment 2 comparing the F1-score (a) and mean F1-score (b) between our threshold-based lexicon approach and machine learning methods.

Similarly, the results from Experiment 2 are displayed in Table 6.2 and box plot presented in Figure 6.2 also show that the lexicon-based approach achieved the highest F1-score of 0.3997 and F1-mean of 0.5266, outperforming the machine learning methods in terms of F1-score. However, machine learning models, including SVM, MultiNB, and LogReg, exhibit lower F1-scores ranging from 0.2759 to 0.3382 and an F1-mean ranging from 0.6061 to 0.6345 when trained on the full, unlimited feature set. In this experiment, the difference in the F1-mean between the lexicon approach and machine learning methods is slightly more noticeable compared to Experiment 1. This shows that Experiment 2 performed better for the Non-Severe class, and the F1-mean

for the Non-Severe is much better than Experiment 1. Moreover, the computation time for classification (tc) shows that SVM took the longest, and in terms of computation time for learning our threshold-based approach took the longest, while MultiNB had the shortest computation time for both learning and classification. The lexicon approach was faster than SVM but not faster than MultiNB and LogReg in terms of classifier computation time. It is crucial to note that the computational cost of the lexicon-based method is primarily incurred during the one-time creation of the Severe and Non-Severe lexicons. After this step, the classification process can be performed efficiently by matching the bug report words with the pre-computed lexicons.

Table 6.3: Results of Experiment 3 as averages of 10 iterations.

	F1-score	F1-mean	TP	FP	TN	FN
lexicon_THR	**0.4174**	0.5281	1070.2	2396	4378.9	582.9
SVM	0.2763	0.5814	315.6	308.4	6459.9	1344.1
MultiNB	0.3652	**0.6204**	504.3	597.3	6171	1155.4
LogReg	0.2878	0.5877	330.4	305.2	6463.1	1329.3

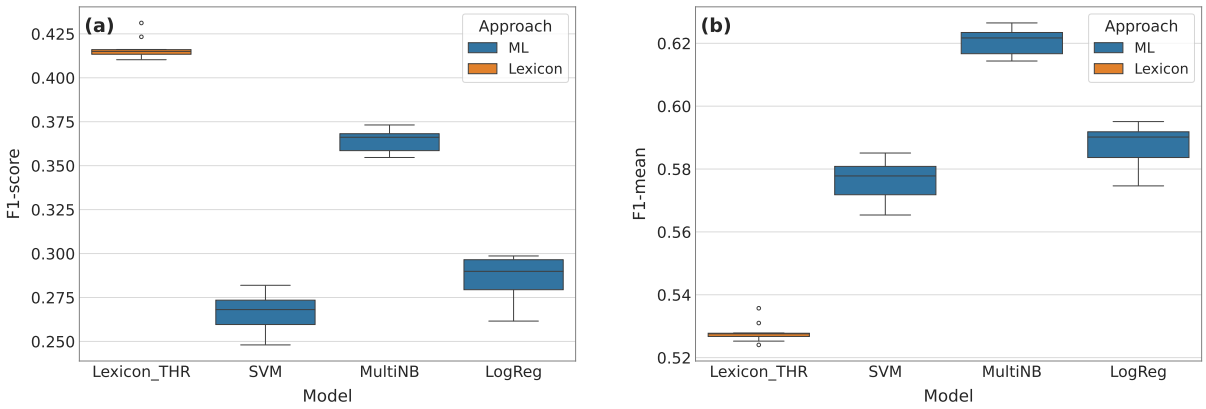

Figure 6.3: Box plots for Experiment 3 comparing the F1-score (a) and mean F1-score (b) between our threshold-based lexicon approach and machine learning methods.

The results of Experiment 3, presented in Table 6.3, and and box plot presented in Figure 6.3 demonstrate that the lexicon-based approach outperformed the machine learning methods, achieving an F1-score of 0.4174. Among the machine learning methods, MultiNB achieved an F1-score of 0.3652, followed by LogReg at 0.2878 and SVM at 0.2763 with infinite features. However, in terms of the mean F1-score, the machine learning methods exhibited better performance, ranging from 0.5814 to 0.6204, compared to the lexicon-based approach, which achieved a mean F1-score of 0.5281.

Table 6.4: Results of Experiment 4 as averages of 10 iterations.

	F1-score	F1-mean	TP	FP	TN	FN
lexicon_THR	0.8925	0.7508	1561.5	284	319.5	92
SVM	0.8874	0.7679	1517	248.4	355.1	136.5
MultiNB	0.8952	**0.7845**	1529.6	233.9	369.6	123.9
LogReg	**0.8969**	0.7786	1553	256.4	347.1	100.5

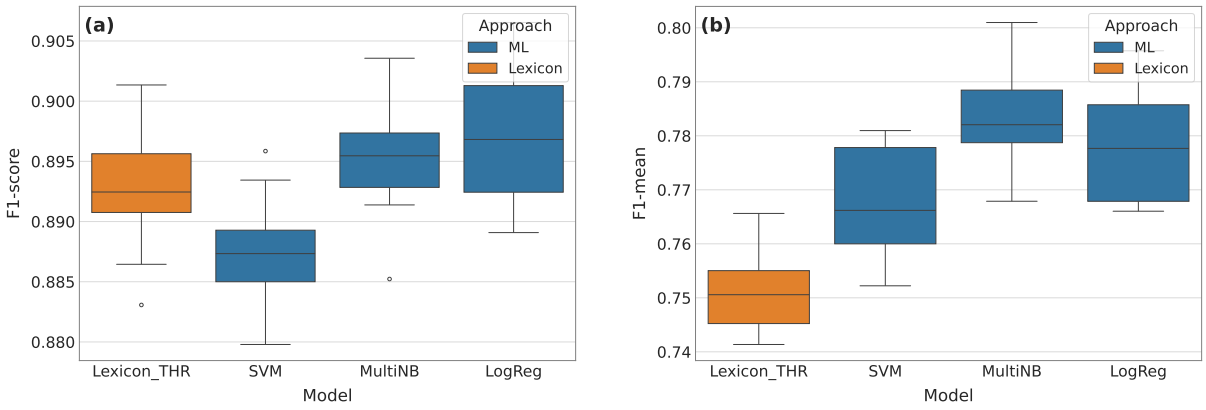

Figure 6.4: Box plots for Experiment 4 comparing the F1-score (a) and mean F1-score (b) between our threshold-based lexicon approach and machine learning methods.

The results of Experiment 4, which includes both the Eclipse and Firefox datasets with the bug status 'normal' filtered to ensure a slightly more balanced data distribution, are presented in Table 6.4 and the and box plot presented in Figure 6.4. The data indicates that MultiNB and LogReg outperform the lexicon-based approach, achieving F1-scores of 0.8952 and 0.8969, respectively. However, the lexicon-based approach, with an F1-score of 0.8925, shows only a relatively small difference in performance compared to the machine learning methods. In terms of the mean F1-score, the machine learning methods exhibit a range of 0.7679 to 0.7845, while the lexicon-based approach achieves a slightly lower mean F1-score of 0.7508.

6.2 Static Lexicons Results

According to the F1-score for the Severe category as the most relevant performance measure in our setting, the lexicon approach looked promising compared to the machine learning methods. This led us to conduct cross dataset experiments to create static lexicons for software bug severity classification. Here, we present the results of the experiments conducted for static lexicons. In this study, two datasets—Firefox and Eclipse—were employed to evaluate various approaches for predicting software bug severity. The methodologies examined included threshold-based lexicon generation, linear SVM coefficient-based lexicon generation, utilization of a publicly available lexicon lexicon_BL, VADER, and machine learning methods. The results of these approaches in a cross-dataset experimental setup, as well as existing lexicon experiments for both the Eclipse and Firefox datasets, are presented in Table 6.5 and Table 6.6 and the results shown in the tables are visually depicted in Figure 6.6 and Figure 6.7 showing graphical representation of F1-score and F1-mean for different approaches tested on the same dataset.

The results of the cross-dataset experiments, where the Firefox dataset was used to create the lexicon and the Eclipse dataset was used for testing, are presented in Table 6.5. The results show that the lexicon_BL method achieved the highest performance with an F1-score of 0.3546, closely followed by lexicon_SVM with an F1-score of 0.3364. In contrast, the lexicon_THR approach performed comparatively lower, achieving an F1-score of 0.2304. When comparing the threshold-based and linear SVM-based lexicon approaches with existing lexicon methods(lexicon_BL, Vader), it is evident that both lexicon_BL and lexicon_SVM outperformed lexicon_THR. Additionally, Vader demonstrated competitive performance with an F1-score of 0.2831.

In comparison to the machine learning methods, all lexicon-based approaches outperformed SVM (F1-score: 0.0615), MultiNB (F1-score: 0.119), and LogReg (F1-score: 0.017). However,

Table 6.5: Comparative evaluation of different approaches on the Eclipse dataset. Results include averages of 10 iterations for cross-dataset experiments (static lexicons created using Firefox data), single-iteration results for lexicon_BL and Vader, and untuned machine learning methods. All approaches were tested on the same Eclipse dataset to enable direct comparison.

	F1-score	F1-mean	TP	FP	TN	FN
lexicon_THR_F	0.2304	0.4137	1218.5	2687.5	18150.5	5336.5
lexicon_SVM_F	0.3364	0.4734	2779.5	7181.5	13656.5	3775.5
lexicon_BL	**0.3546**	0.4809	4116	12539	8299	2439
Vader	0.2831	**0.5144**	2031	5758	15080	4524
SVM	0.0615	0.4599	221.3	410.8	20427.2	6333.7
MultiNB	0.1198	0.4865	469	802	20036	6086
LogReg	0.0170	0.4399	57	90	20748	6498

Table 6.6: Comparative evaluation of different approaches on the Firefox dataset. Results include averages of 10 iterations for cross-dataset experiments (static lexicons created using Eclipse data), single-iteration results for lexicon_BL and Vader, and untuned machine learning methods. All approaches were tested on the same Firefox dataset to enable direct comparison.

	F1-score	F1-mean	TP	FP	TN	FN
lexicon_THR_E	**0.2559**	0.4506	592.1	2373.3	10712.7	1066.9
lexicon_SVM_E	0.2469	0.4462	678.1	3151.7	9934.3	980.9
lexicon_BL	0.2141	0.4297	1143	7842	5244	516
Vader	0.2218	0.5112	667	3688	9398	992
SVM	0.1576	0.5374	208	771.9	12314.1	1451
MultiNB	0.1718	0.5465	220	681	12405	1439
LogReg	0.1746	**0.5510**	208	515	12571	1451

in terms of the F1-mean, Vader achieved the highest F1-mean of 0.5144, while the threshold-based lexicon approach achieved the lowest F1-mean of 0.4137 and the machine learning methods achieved F1-mean better than lexicon_THR_F. These findings indicate that lexicon-based methods, particularly lexicon_SVM_F and lexicon_BL, are more effective for predicting software bug severity compared to lexicon_THR_F and the machine learning models evaluated in this study.

The results of the study, where the lexicon was created using the Eclipse dataset and tested on the Firefox dataset, are presented in Table 6.6. On the Firefox dataset, the evaluation of various classification methods based on their F1-scores reveals that all approaches achieved relatively low F1-scores. However, lexicon_THR performed better than the others, achieving an F1-score of 0.2555. Among the lexicon-based methods, lexicon_SVM and Vader also demonstrated competitive performance, with F1-scores of 0.2469 and 0.2218, respectively. In contrast, the lexicon_BL approach achieved a slightly lower F1-score of 0.2141.

When compared to the machine learning methods, the lexicon-based approaches outperformed the machine learning models in this experimental setting. Specifically, the SVM approach achieved an F1-score of 0.1576, while MultiNB and LogReg achieved F1-scores of 0.1718 and 0.1746, respectively. However, in terms of the F1-mean, the machine learning methods performed better than all lexicon-based approaches, as evident in the table. This suggests that, in this experimental setting, the machine learning methods were more effective in classifying the Non-Severe class.

6.2.1 Qualitative Analysis

We conducted a qualitative analysis for three lexicons: the existing lexicon (BingLiu), the final merged threshold-based lexicon created on cross dataset, and the final merged linear SVM-based lexicon, also created on a cross dataset. This qualitative assessment aims to analyze how the

choice of words in these lexicons relates to the severity of software bugs. In some cases, a negative term may be classified in the Non-Severe category, or conversely, a positive term may be assigned to the Severe category due to the specific vocabulary used by the individual bug reporter. For this purpose, the top 25 most frequently occurring words from the above approaches are enlisted below to better understand their relevance and categorization within the context of software bugs and their severity.

Intersection of the lexicon_BL lexicon with Eclipse dataset

The purpose of the qualitative analysis regarding lexical intersections of lexicon_BL and the summary text column in the software bug report for Eclipse is to explore the relationship between the languages users use in bug reports and the corresponding terms in the opinion lexicon. The aim is to identify patterns that may help categorize the bug severity correctly. Interestingly, lexicon_BL performed comparatively better on the Eclipse dataset, making it interesting for a qualitative comparison in this case. The methodology for determining the intersection is an exact match of words from lexicon_BL and the words from the summary column in the bug report. The most frequently intersecting examples are provided below:

Intersection of Negative Words from `lexicon_BL` with the *Summary* Column of Eclipse Dataset

error: 1504, fail: 836, miss: 701, incorrect: 469, break: 391, invalid: 386, wrong: 364, problem: 324, unable: 277, issue: 224, hang: 198, disable: 195, deadlock: 193, object: 189, ignore: 184, bug: 166, incorrectly: 160, failure: 144, static: 131, slow: 113, leak: 105, crash: 98, inconsistent: 95, false: 86, failures: 82, regression: 79, bad: 78,...

Intersection of Positive Words from `lexicon_BL` with the *Summary* Column of Eclipse Dataset

work: 792, support: 551, dynamic: 269, properly: 236, correctly: 234, clean: 195, available: 143, improve: 138, refresh: 121, correct: 88, right: 83, better: 77, respect: 68, top: 63, flexible: 57, lead: 56, improvements: 51, like: 47, progress: 44, consistent: 40, proper: 40, clear: 38, well: 37, prompt: 34, contribution: 30, ...

Category-wise List of Intersecting Negative Words from `lexicon_BL` with the *Summary* Column of Eclipse Dataset

Words only in Severe category: terrible, erase, emergency, burn, inopportune, dubious, radically, unsafe, fake, incompatability, painful, erratically, clueless, cripple, discriminate, degrade, sting.
Words only in Non-Severe category: harden, rogue, downcast, exclusion, haywire, suspect, alarm, ambiguous, harsh, annoy, slave, gutter, regress, fat, downgrade, insensitively, unhelpful, mistake, scratch, sporadic, restriction, lag, undesirable, harm, indistinguishable, ...

Category-wise List of Intersecting Positive Words from `lexicon_BL` with the *Summary* Column of Eclipse Dataset

Words only in Severe category: responsive, awesome, warm, portable, overtake, rapid, intelligent, guidance, protection, reliably.
Words only in Non-Severe category: classic, free, realistic, effective, precise, easier, perfectly, modest, straightforward, endorse, enhance, smarter, steady, excel, eager, logical, guarantee, stability, favor, nicely, pros, facilitate, suitable, selective, appropriate, ...

The intersecting words from the positive and negative wordlists of lexicon_BL and the bug report fall into both Severe and Non-Severe categories. In the Severe category, words like 'terrible', 'emergency', 'unsafe', 'destroy', and 'painful' express strong negative sentiments, indicating issues that could impact user experience. This shows that the lexicon is effective in the context of software bug severity classification. From the intersection of positive words in the Bing Liu lexicon with the Eclipse dataset, words like 'responsive', 'guidance', 'reliably', and 'protection' appear in the positive wordlist of Bing Liu lexicon but intersect with Severe bugs in the report, indicating minor issues or suggestions. Interestingly, words like 'mistake', 'restriction', 'lag', 'ambiguous', 'harsh', and 'annoy' from Bing Liu lexicon's negative words intersect with Non-Severe bugs in the Eclipse bug report. This also depends on the quality of the bug report, though these words should appear with the Severe bugs in the report because it clearly shows user frustration and urgency.

Intersection of the threshold-based custom lexicon with Eclipse dataset

This section presents the qualitative analysis conducted on the final lexicon generated using our threshold-based approach. In the following section, we first display the top 25 intersecting words and their counts from the Severe and Non-Severe categories of the threshold-based lexicon with the complete dataset of Eclipse. Next, we show the intersecting words from the Severe and Non-Severe categories within the Eclipse dataset. In a threshold-based lexicon, a word could appear in both categories, as the lexicon is created based on various combinations of thresholds. The objective of categorizing the words is to identify unique words from our final lexicon that originally appeared in either the Severe or Non-Severe category.

Intersection of Severe Words from `lexicon_THR` with the *Summary* Column of Eclipse Dataset

not: 4664, project: 3321, file: 2320, web: 2208, xml: 2045, server: 1835, validation: 1694, error: 1504, add: 1270, wizard: 1092, update: 1072, j: 1071, create: 1024, new: 1023, ee: 1002, service: 929, change: 877, build: 860, fail: 836, ear: 831, view: 817, wtp: 810, work: 792, class: 773, import: 709, jar: 673, ...

Intersection of Non-Severe Words from `lexicon_THR` with the *Summary* Column of Eclipse Dataset

not: 4664, project: 3321, file: 2320, web: 2208, xml: 2045, editor: 1828, validation: 1694, error: 1504, add: 1270, java: 1255, org: 1196, use: 1145, page: 1127, need: 1114, update: 1072, j: 1071, new: 1023, ee: 1002, test: 944, content: 930, service: 929, type: 888, change: 877, remove: 860, fail: 836, view: 817, ...

Category-wise Intersecting Severe words from `lexicon_THR` with the *Summary* Column of Eclipse Dataset

Words only in Severe category: erase, kernel, shift, acknowledge, creat, reliably, configurationerror, severty, acknowledge, linkageerror, iterate, firewalls, burn, flood, destabilize, indexoutofboundsexception, suspendvalidation, executionexception, latet, errrors, malforming, deadload, unsupportedencoding, correctness, wake, duplication, sting, unaccessable, asynch, errretrt, broadcast, swterror ...
Words only in Non-Severe category: green, distinguish, backup, spam, sucessfully, resurrect, dual, daily, multiply, forward, cyrillic, lan, nightly, june, program, ugly, quite, incompatibility, mistyped, pause, authorization, limitation, quit, filesize, death, invisible, changeable, potentially, temporarily, dismiss, inaccurate, vanish, slower ...

Category-wise Intersecting Non-Severe words from `lexicon_THR` with the *Summary* Column of Eclipse Dataset

Words only in Severe category: government, backout, blocker, flood, unsafe, unaccessable, spawn, degrade, erratically, cripple, recycle, reboot, erratically, painful, uninitialized, intervals, weekly, installations, portable, implications, proceed, additionally, responsive, repetitive mr, cup, life, ...
Words only in Non-Severe category: logwarning, connectexception, abnormal, backout, startserver, validateentries, removelistener, illegalargumentsexception, reboot, accelerator, val, pasted, outdated, stacktrace, spuriously, synching, inconsistancy, misreports, internalresolvetype, validatepage, inferencing, restarts, trivially, interterface, ...

Our qualitative analysis for the threshold-based lexicon shows that, in the Severe category, words like 'configurationerror', 'linkageerror', 'errors', 'unaccessible', 'severity', and 'deadload' indicate major issues and correctly appear in the Severe category in the final lexicon. However, words like 'dual', 'limitation', 'mistyped', and 'temporarily', for example, indicate that these words do not correctly belong in the Severe category. Conversely, the Non-Severe category of our lexicon includes words like 'logwarning', 'inconsistency', 'validatepage', and 'trivially' which indicate minor issues and correctly appear in the Non-Severe category in the lexicon. Moreover, words like 'blocker', 'illegalargumentexception', and 'painful' have appeared in the Non-Severe category of the lexicon, indicating misclassification. However, some words like 'wake', 'June' and 'government' do not show any urgency or indication of bugs, suggesting that these should not appear in the lexicon. The frequent appearance of 'not' suggests the need to consider negating words.

Intersection of the linear SVM-based custom lexicon with Eclipse dataset

In this section we explain the insights from our qualitative analysis from the intersection of linear SVM-based lexicon with the Eclipse dataset. The following tables show the top 25 words for each analysis section.

Intersection of Severe Words from `lexicon_SVM` with the *Summary* Column of Eclipse Dataset

not: 4664, project: 3321, web: 2208, xml: 2045, server: 1835, eclipse: 1701, validation: 1694, error: 1504, add: 1270, jsp: 1233, org: 1196, npe: 1141, need: 1114, wizard: 1092, create: 1024, new: 1023, ee: 1002, service: 929, type: 888, build: 860, fail: 836, ear: 831, wst: 828, wsdl: 821, wtp: 810, work: 792, import: 709, miss: 701, jar: 673, ejb: 662, open: 650, exception: 613, cannot: 598, module: 587, ...

Intersection of Non-Severe Words from `lexicon_SVM` with the *Summary* Column of Eclipse Dataset

project: 3321, file: 2320, server: 1835, editor: 1828, validation: 1694, error: 1504, add: 1270, java: 1255, org: 1196, use: 1145, page: 1127, need: 1114, wizard: 1092, update: 1072, new: 1023, ee: 1002, test: 944, content: 930, service: 929, type: 888, change: 877, remove: 860, view: 817, name: 812, ui: 805, class: 773, miss: 701, jar: 673, tag: 672, attribute: 662, show: 650, default: 640, ...

Category-wise Intersecting Severe words from `lexicon_SVM` with the *Summary* Column of Eclipse Dataset

Words only in Severe category: configurationerror, degrade, terrible, linkageerror, indexoutofboundexception, erratically, cripple, emergency, unpublish, destabilize, npex, flood, expection,validation, add, create, import, module, update, restart, build, debug burn, billions, displose, clueess, concatenated,...
Words only in Non-Severe category: inaccessible, mistyped, incompatibility, inaccurate, limitation, unexplainable, invisible, dissapear, typos, insufficient, inconsistency, misinterpret, abnormally, phantom, borked, modifiying, constructure, parcing, collision, death, 'admin', 'orientation', 'modifiying', 'percent', 'stretch', 'slower' ...

Category-wise Intersecting Non-Severe words from `lexicon_SVM` with the *Summary* Column of Eclipse Dataset

Words only in Severe category: blocker, unaccessable, flood, unsafe, degrade, cripple, erratically, deserialization, unintended, recycle, uninitialized, painful, fake, reboot, experience, implications, uninitialized, unintended, degrade, greek, capitan, recently, unix, popups, reappear, inconsistancies, discriminate
Words only in Non-Severe category: borked, unsuccessful, illegible, failed, conform, restartimpl, recommend, arithmeticexception, malformedtreeexception, exception, invalidstate, overclock, compilerfailure, typeerror, nodenameerror, referenceerror, missingmethodexception, dependencyerror, compilation, faultyexecution,versionchecker, abstractdocumentloader, entitygroups, detach, plaintext, ...

From our analysis of the intersection of the linear SVM-based custom lexicon with the Eclipse dataset, it was observed that in the Severe category from the linear SVM-based final lexicon, such as 'configurationerror', 'terrible', 'linkage error', 'crashing', 'emergency', and 'destabilize' show critical issues that could lead to system failure and user frustration. This indicates that these words correctly appear in the Severe category of the lexicon. On the other hand, the Non-Severe category includes words like 'typeerror', 'detach', 'restartimpl', and 'nodenameerror' which indicate minor inconveniences and are correctly appear in the Non-Severe category in the final lexicon. However, words like 'blocker', 'flood', and 'unsafe' seem that it should not appear in the Non-Severe category of the lexicon; moreover, these words have intersections with Severe bugs in the bug report, which also suggests that these words express user inconvenience and should be classified as Severe.

Intersection of lexicon_BL with Firefox dataset

This section presents the qualitative analysis of Bingliu's lexicons used in our study with the Firefox dataset.

Intersection of Negative Words from `lexicon_BL` with the *Summary* Column of Firefox Dataset

error: 605, fail: 511, intermittent: 413, alarm: 362, wrong: 345, bug: 300, crash: 293, drag: 232, miss: 227, break: 218, disable: 208, incorrect: 162, unable: 126, lose: 110, issue: 108, problem: 104, hang: 95, unexpected: 84, freeze: 83, slow: 80, dark: 76, incorrectly: 75, failure: 72, errors: 72, ignore: 67, ...

Intersection of Positive Words from `lexicon_BL` with the *Summary* Column of Firefox Dataset

work: 620, right: 173, correctly: 149, refresh: 139, properly: 134, top: 127, support: 120, prompt: 91, clear: 90, win: 81, correct: 81, like: 71, available: 66, protection: 47, progress: 37, respect: 37, accessible: 34, improve: 34, better: 32, exceed: 32, free: 31, well: 30, lead: 27, master: 26, compatible: 25, safe: 22, ...

Category-wise List of Intersecting Negative Words from `lexicon_BL` with the *Summary* Column of Firefox Dataset

Words only in Severe category: miscalculate, limitation, jitters, death, concern, drain, obtuse, unstable, flakey, degradation.
Words only in Non-Severe category: dangerous, dead, incomplete, virus, unreadable, painful, interrupt, unavailable, malicious, Severe, mess, idle, unresponsive, sticky, broken, freeze, ineffective, slower, unexpected, glitch, contradictory, vice, shake, fuzzy, junk

Category-wise List of Intersecting Positive Words from `lexicon_BL` with the *Summary* Column of Firefox Dataset

Words only in Severe category: important, advantage, guarantee, gutsy, accurately.
Words only in Non-Severe category: smooth, bright, recommendation, readable, adaptive, fine, fast, revive, capable, usable, reachable, effective, recover, fluent, quiet, attractive, fancy, improvement, comfort, eagerly, precisely, good, prompt, safely, simplify, support, ...

Our analysis on the intersection of words from Bing Liu's positive and negative word lists with the Firefox dataset shows that, in the Severe category, words like 'improper', 'miscalculate' and 'drain' express negative sentiments, indicating issues that could impact user experience. This suggests that Bing Liu's lexicon performs well in the context of software bug severity classification. However, some intersecting words from the negative words in the lexicon also appear in Non-Severe bug reports. For example, words like 'dangerous', 'dead', and 'virus' are strong negative words, but they appear with the Non-Severe bugs in the bug reports. This indicates the quality of a bug reports, what type of vocabulary used by different authors. In the Non-Severe category, words like 'improvement', 'simplify', 'revive', and 'support' indicate minor issues and suggests enhancements, accurately categorizing them as Non-Severe. Moreover, words like 'shake', 'quiet', 'sticky', and 'funny' are not very helpful in the context of bug severity classification.

Intersection of threshold-based custom lexicon with Firefox dataset

This section presents the qualitative analysis of the intersection of our threshold-based custom lexicon with the Firefox dataset.

Intersection of Severe Words from `lexicon_THR` with the *Summary* Column of Firefox Dataset

calendar: 2936, not: 2728, event: 1430, lightning: 1047, events: 910, new: 884, view: 786, show: 773, time: 735, file: 732, task: 723, display: 683, work: 620, error: 605, change: 564, date: 538, add: 537, fail: 511, day: 498, update: 495, sunbird: 430, thunderbird: 426, intermittent: 413, cannot: 401, google: 378, windows: 377, start: 369, ...

Intersection of Non-Severe Words from `lexicon_THR` with the *Summary* Column of Firefox Dataset

calendar: 2936, not: 2728, event: 1430, browser: 1148, tab: 942, firefox: 885, new: 884, open: 842, view: 786, show: 773, test: 752, time: 735, file: 732, task: 723, js: 718, use: 692, display: 683, window: 665, button: 661, page: 647, work: 620, error: 605, change: 564, menu: 559, dialog: 541, ...

Category-wise Intersecting Severe words from `lexicon_THR` with the *Summary* Column of Firefox Dataset

Words only in Severe category: icalendars, recurrent, quality, workspace, adata, kinds, edu, academia, runner, funny, bookmarkfolder, renderviewpage, ftp, docaccessibleparent, newtree, drain, gpos, worker, gtkparent, cpp, improper, activation, inspect, windowsjumplists, mozstorageservice ...
Words only in Non-Severe category: eagerly, regression, revert, delay, problem, quit, unpredictable, break, malfunction, degrade, suspend, configure, initiate, access, login, intermittent, inaccurate, install, update, misconfigure, notify, plugin, funny, team, ebay, activate ...

Category-wise Intersecting Non-Severe words from `lexicon_THR` with the *Summary* Column of Firefox Dataset

Words only in Severe category: funny, improper, runner, inspect, concern, quite, towards, modifiying, installable, someday, comparator, encodings, death, enumeration, linefeeds, sucessfully, mistyped, allocation, preselected, runner, nod, attemping, concurrency, strategy, checkin,...
Words only in Non-Severe category: exhaustion, block, unpredictable, spawn, unfinished, sanitizeonshutdown, forever, interaction, error, reset, configure, improperly, resizable, negotiation, hoverable, noticeable, interact, units, newtab, context, paint, forever, noticeable, organization, package, ...

Our analysis on the intersection of words from the threshold-based final lexicon and the Firefox dataset provides the following insights about the final lexicon. Firstly, the frequent appearance of 'not' shows that it is important to consider negation. Secondly, the Severe category of the lexicon includes words like 'drain' and 'improper', which indicate major issues and correctly appear in the Severe category. However, words such as 'regression', 'revert', 'delay', 'quit', and 'unpredictable' also appear in the Severe category of the final lexicon, which is correct. However, the intersection of words shows that these words have appeared with the severity level 'Non-Severe' in the bug reports, indicating that these words can help in predicting bug severity but are not urgent. Moreover, some words, such as 'cpp' 'update' and 'funny' lack context and do not help predict a software bug. This shows that these words in the lexicon are not helpful in bug classification. Conversely, the Non-Severe category includes words like 'reset', 'configure', 'improperly', 'resizable', 'negotiation', and 'noticeable', which indicate minor issues and are

correctly classified. However, words like 'exhaustion', 'block', 'unpredictable', and 'error' have appeared in the Non-Severe category in the final lexicon. These words show more urgency and should appear in the Severe category of the final lexicon.

Intersection of linear SVM-based custom lexicon with Firefox dataset

This section presents the qualitative analysis of the intersection of our linear SVM-based custom lexicon with the Firefox dataset.

Intersection of Severe Words from `lexicon_SVM` with the *Summary* Column of Firefox Dataset

calendar: 2936, not: 2728, event: 1430, lightning: 1047, new: 884, open: 842, time: 735, task: 723, js: 718, display: 683, work: 620, error: 605, add: 537, fail: 511, sunbird: 430, thunderbird: 426, intermittent: 413, cannot: 401, google: 378, set: 375, start: 369, alarm: 362, create: 361, wrong: 345, save: 338, delete: 314, bug: 306, ...

Intersection of Non-Severe Words from `lexicon_SVM` with the *Summary* Column of Firefox Dataset

calendar: 2936, browser: 1148, tab: 942, firefox: 885, new: 884, view: 786, show: 773, test: 752, file: 732, task: 723, use: 692, display: 683, window: 665, button: 661, page: 647, error: 605, change: 564, menu: 559, dialog: 541, date: 538, add: 537, day: 498, update: 495, bar: 490, click: 466, ...

Category-wise Intersecting Severe words from `lexicon_SVM` with the *Summary* Column of Firefox Dataset

Words only in Severe category: ftp, adata, newtree, inspect, renderviewpage, worker, gtkparent, cpp, improper, nsprintdialoggtk, gpos, docaccessibleparent, mozstorageservice, kinds, edu, workspace, quality, bookmarkfolder, academia, drain, activation, runner, ...
Words only in Non-Severe category: malformed, inability, dirty, free, break, stage, searchable, plain, snapshot, faculty, unstable, python, undefined, transport, end, nonfunctional, sitecore, adjustment, encrypt, error, tether, sparse, mean, me, gallery, ...

Category-wise Intersecting Non-Severe words from `lexicon_SVM` with the *Summary* Column of Firefox Dataset

Words only in Severe category: structure, consumed, important, write, improper
Words only in Non-Severe category: particular, silently, ssh, welcome, slow, improvement, packaged, fxa, log, affecting, stick, accesskeys, highlighting, instantly, yet, encountered, entry, merged, arrange, violation, bugged, across, msg, antivirus, suggestion, modern

The insights of our qualitative analysis of the linear SVM-based lexicon intersection with the Firefox dataset, shows that in the Severe category, words like 'drained', and 'improper' indicate major issues and correctly appear in the Severe category of the lexicon. However, some words, like 'ftp', 'activation', and 'bookmarkfolder' have no context and do not help in the classification of a bug, and such words are safe to remove from the final lexicon. On the other hand, the Non-Severe category includes words like 'improvement', 'encountered', and 'minimize', which indicate minor issues and correctly appear in the Non-Severe category of the lexicon. Notably, words like 'bugged' and 'violation' have appeared in the Non-Severe but imply critical problems, which shows that they are misclassified.

6.3 Considering Negation

In the scope of of our qualitative analysis of the lexicons created using threshold-based and linear SVM coefficient-based methods, we found that the word 'not' appeared very frequently. In the preprocessing steps, we chose to retain the word 'not' while removing other stopwords, as it holds significant value for our analysis. Following our quantitative and qualitative analyses, we addressed the issue of negation during the preprocessing phase by concatenating 'not' with the subsequent word (e.g., 'not_reliable').

Frequent words appearance in both datasets before and after handling negation

The table below displays the most frequent words in the summary column of the datasets, before and after handling negation.

Top 25 frequent appearing words in Eclipse dataset before and after handling negation

Before: not: 4664, project: 3321, file: 2320, web: 2208, xml: 2045, server: 1835, editor: 1828, eclipse: 1701, validation: 1694, error: 1504, add: 1270, java: 1255, jsp: 1233, org: 1196, need: 1114, npe: 1141, j: 1071, create: 1024, ee: 1002, service: 929, fail: 836, wtp: 810, ear: 831, import: 709...
After: project: 3319, file: 2320, web: 2207, xml: 2045, server: 1834, editor: 1828, eclipse: 1701, validation: 1694, error: 1503, java: 1255, add: 1212, jsp: 1233, org: 1196, need: 1098, npe: 1141, j: 1071, create: 979, ee: 1002, ear: 831, service: 929, fail: 834, ... not_add: 58, ... not_create: 45, ... not_need: 16, ... not_fail: 2, not_project: 2, ... not_server: 1, not_error: 1, not_web: 1

Top 25 frequent appearing words in Firefox dataset before and after handling negation

Before: calendar: 2936, not: 2728, event: 1430, browser: 1148, lightning: 1047, tab: 942, firefox: 885, new: 884, open: 842, view: 786, show: 773, test: 752, time: 735, task: 734, file: 732, js: 718, display: 683, use: 692, button: 661, window: 665, page: 647, work: 620, error: 605, change: 564, menu: 559, ...
After: calendar: 2935, event: 1429, browser: 1148, lightning: 1045, tab: 942, firefox: 885, new: 884, open: 805, view: 786, time: 734, task: 732, file: 732, error: 605, add: 521, fail: 510, update: 430, thunderbird: 426, ... work: 359, ... not_work: 261, ... not_show: 147, ... not_update: 65,... message: 48,... not_open: 37,... not_add: 16, ... not_calendar: 1,... not_event: 1

This experimentation was conducted for all the lexicon approaches and machine learning methods, and the results are shown from Table 6.7 - Table 6.8. Our hypothesis proposed that handling the negation would improve performance across all experiments, particularly in F1-scores. However, the results contradicted our expectations. For established lexicons like VADER and Bing Liu lexicon performance metrics remained unchanged after negation handling in both experimental settings. For threshold-based lexicon methods and ML methods, there was a slight improvement in F1-score in the second experimental setting. However, linear SVM-based lexicon approach showed no performance gains after negation handling, it has even performed worst. These findings suggest that incorporating negation handling did not produce the anticipated improvements in bug severity classification accuracy.

Below is an example of a preprocessed summary text, both before and after negation handling, along with the most frequently negated words identified in the final lexicon after negation handling.

```
Example:
Summary: Firefox is not working when opening multiple tabs.
Before negation: ['firefox', 'not', 'work', 'open', 'multiple', 'tab']
After negation: ['firefox', 'not_work', 'open', 'multiple', 'tab']
```

Frequent Negated Words appearance in the threshold-based lexicon

Severe: not_auth ratio: 1.0, not_build ratio: 1.0, not_changeable ratio: 1.0, not_choke ratio: 1.0, ... not_launch ratio: 0.75, not_check ratio: 0.666, not_convert ratio: 0.6666, not_install ratio: 0.625, not_cause ratio: 0.5, not_consistently ratio: 0.5, ... not_start ratio: 0.470, not_fire ratio: 0.428, not_register ratio: 0.428, not_create ratio: 0.416, not_accept ratio: 0.333, not_editable ratio: 0.333, not_enough ratio: 0.333, ... not_find ratio: 0.266, not_load ratio: 0.258, not_actually ratio: 0.25, not_adjust ratio: 0.25, not_available ratio: 0.233, not_initialize ratio: 0.222, not_handle ratio: 0.214, ...
Non-Severe: not_persist ratio: 1.0, not_persistent ratio: 1.0, not_pick ratio: 1.0, not_pin ratio: 1.0, not_preloaded ratio: 1.0, not_present ratio: 1.0, not_preserve ratio: 1.0, not_press ratio: 1.0, not_proceed ratio: 1.0, not_always ratio: 0.956, not_correctly ratio: 0.952, not_change ratio: 0.951, not_respect ratio: 0.947, not_correct ratio: 0.944, not_render ratio: 0.937, not_match ratio: 0.9333, not_define ratio: 0.931, not_delete ratio: 0.916, not_restore ratio: 0.916, not_allow ratio: 0.913, not_use ratio: 0.903, not_get ratio: 0.9, not_visible ratio: 0.9, ...

6.3.1 Results After Handling Negation

The results of the experiments conducted for handling negation are presented in Tables 6.7–6.8. Furthermore, the results shown in the tables are visually depicted in Figures 6.6 and Figure 6.7, which provide a graphical representation that compares different approaches before handling the negation and after, in terms of the F1-score and the F1-mean. The results from

Table 6.7: Results showing averages of 10 iterations of experiments where static lexicon was created with Firefox dataset and tested with Eclipse dataset after handling negation

	F1-score	F1-mean	TP	FP	TN	FN
lexicon_THR_F	0.2181	0.4125	1094.9	2372.4	18465.6	5460.1
lexicon_SVM_F	0.3135	0.4611	2388.7	6282.8	14555.2	4166.3
lexicon_BL	**0.3517**	0.4790	4258	12780	8058	2297
Vader	0.2831	**0.5144**	2031	5758	15080	4524
SVM	0.0640	0.4606	232.2	461.5	20376.5	6322.8
MultiNB	0.1132	0.4828	442	812	20026	6113
LogReg	0.016	0.4396	55	94	20744	6500

the study in which the lexicon was created with the Firefox dataset and tested on the Eclipse dataset and after handling negation are shown in Table 6.7. The results show the lexicon_BL approach consistently achieved better performance both before and after negation handling, followed by Vader. The lexicon_THR and lexicon_SVM approach showed a slight decline in performance after negation handling, while the machine learning models generally had lower F1-scores in both scenarios, with only minor changes after negation handling. This indicates that the lexicon-based methods, especially lexicon_BL and lexicon_SVM, performed well in terms of F1-score after negation handling. The results from experiments where lexicon was created on Eclipse dataset and tested with Firefox dataset and after handling negation are displayed in table 6.8, and in Figure 6.7, the results show that after considering negation the F1-score for lexicon_THR improved slightly, while the F1-score for lexicon_SVM, lexicon_BL and Vader slightly declined with few decimals but almost remained unchanged. Furthermore, to compare

Table 6.8: Results showing averages of 10 iterations of experiments where static lexicon was created with Eclipse dataset and tested with Firefox dataset after handling negation.

	F1-score	F1-mean	TP	FP	TN	FN
lexicon_THR_E	**0.2612**	0.4533	604.4	2361.9	10724.1	1054.6
lexicon_SVM_E	0.2437	0.4443	625	2841.2	10244.8	1034
lexicon_BL	0.2138	0.4296	1154	7981	5105	505
Vader	0.2218	0.5112	667	3688	9398	992
SVM	0.18981	**0.5532**	261	830.1	12255.9	1398
MultiNB	0.1747	0.5480	224	681	12405	1435
LogReg	0.1675	0.5472	199	518	12568	1460

the machine learning methods, before and after negation, the F1-score for SVM and MultiNB improved slightly after negation.

6.4 LLM Zero-Shot Evaluation

Large language models are a state-of-the-art approach for text classification tasks. They do, however, require significant computing resources. In our work, we were primarily interested in comparison of lexicon approaches with the conventional machine learning approaches, as they are more comparable in terms of resource demands and as their performance was expected to be in between that of the lexicon approaches and the LLMs. It is also not straightforward to evaluate the LLMs and interpret their evaluation results, as the LLMs have most probably used all the public textual datasets and have therefore also observed all the test data. However, we have conducted a simple experiment also with the LLMs to provide insight into what one could expect from a smaller quantized LLM that can be deployed on a local server.

We assessed three quantised LLMs—`gemma3:27b`, `llama3.3:70b`, and `deepseek-r1:14b`—on the Firefox bug-report dataset (14,745 summaries). Each model was queried through a locally hosted API with the following zero-shot prompt:

> *You are a bug-severity classifier. Classify the following bug-report summary as either Severe or Non-Severe. Return only the label.*

The API returned a single label, and any call that failed was excluded from metric computation. All metrics were calculated on the remaining, successfully classified summaries. F1-mean denotes the macro-average of the Severe and Non-Severe F1-scores.

Table 6.9: Results of a zero-shot evaluation of LLMs on the Firefox dataset

	F1-score(Severe)	F1-mean	TP	FP	TN	FN
deepseek-r1:14b	0.1778	0.5354	302	1435	11650	1357
llama3.3:latest	**0.3773**	**0.6359**	769	1747	10769	791
gemma3:27b	0.3287	0.5427	1289	4895	8185	369

Our evaluation of selected LLMs on the Firefox dataset, which contains 14,745 bug reports, demonstrates the potential of LLMs for automated bug severity classification. The llama3.3:latest model showed the best performance, successfully classifying 14,076 out of 14,745 summaries (with 669 cases excluded due to API call failures), and achieved an F1-score of 0.3773 for the Severe class and an F1-mean of 0.6359. The gemma3:27b model provided a strong balance between performance and reliability, with an F1-score of 0.3287 for the Severe class and an F1-mean of 0.5427, successfully classifying 14,738 summaries (only 7 excluded due to API failures). In comparison, the deepseek-r1:14b model processed 14,744 summaries (1 excluded) but had a lower Severe F1-score of 0.1778 and an F1-mean of 0.5354. Both llama3.3 and gemma3:27b

outperformed our lexicon-based approach (F1-score: 0.2612 for the Severe class), showing the usefulness of LLMs for bug severity classification. Experiments on public datasets, however, cannot provide a very strong indication whether the same kind of improvement would be observed also in completely unseen test data.

6.5 Results Summary

Across all the experimental settings, lexicon-based approaches, particularly those using threshold-based and linear SVM-based lexicon generation, demonstrated competitive performance in classifying bug severity, often outperforming traditional machine learning models in terms of F1-score for the Severe class. However, machine learning methods generally performed better for the Non-Severe class, as reflected in the higher F1-mean scores. Classification time of lexicon-based methods was faster than of the SVM but slower than LogReg and MultiNB, which was somewhat unexpected, although code optimization was not a focus in this work. Our developed static lexicons performed well compared to existing lexicons, with results heavily influenced by dataset quality and domain. In particular, even general sentiment lexicons without software-specific vocabulary proved to be surprisingly effective. Handling negation did not yield significant improvements in most cases. Overall, we found the lexicon-based methods to be effective and robust for bug severity classification, especially when training data is limited, while machine learning models may offer advantages in more balanced or data-rich scenarios. In addition, a zero-shot evaluation with LLMs was conducted on the Firefox dataset, with LLMs assessed using the same data and task as our threshold-based approach. The results show that both llama3.3 and gemma3:27b outperformed our method for bug severity classification, while deepseek-r1:14b was less effective for the Severe class. These findings highlight the potential of LLMs for this task, even without additional training.

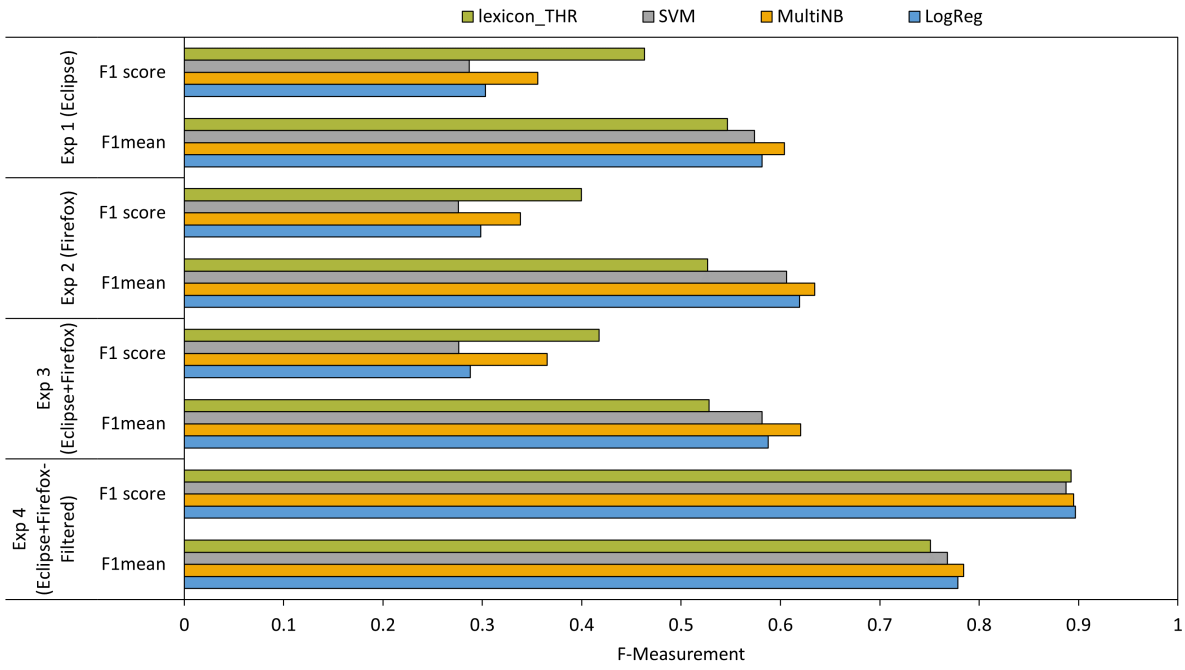

Figure 6.5: Dynamic threshold-based lexicon results in comparison with and machine learning methods on different experimenting settings.

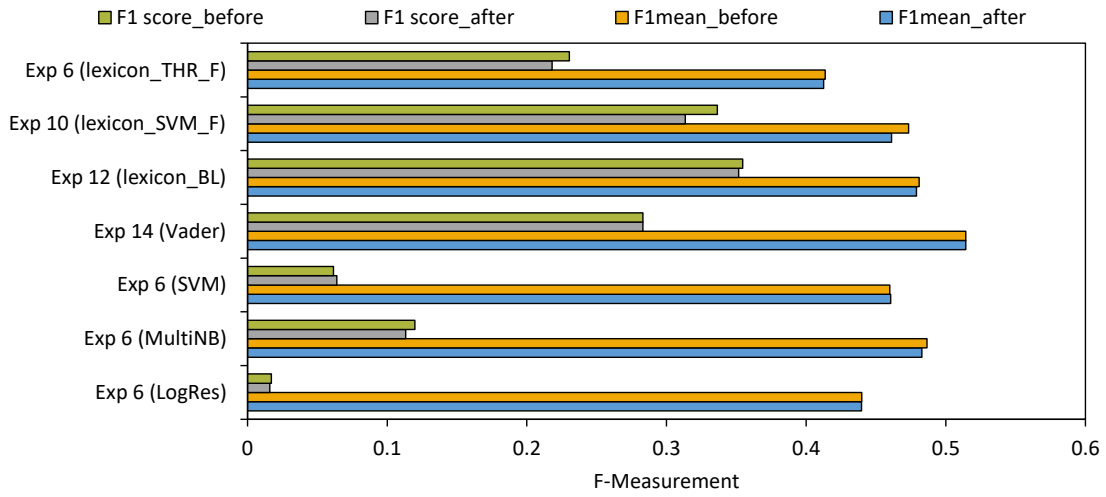

Figure 6.6: Static lexicon results for different approaches tested on the Eclipse dataset before and after handling negation, this shows how different approaches performed before and after handling negation on the same experimental settings.

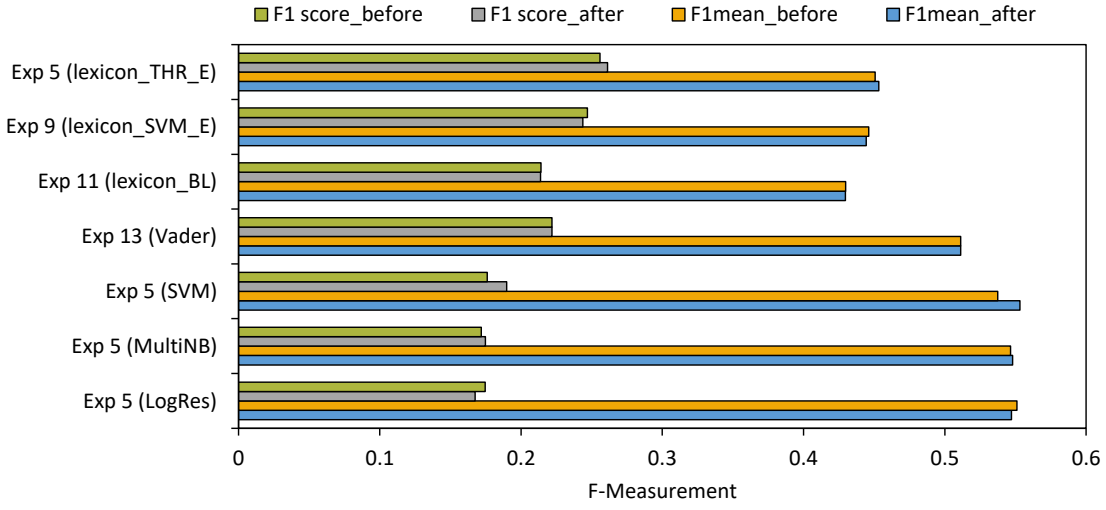

Figure 6.7: Static lexicon results for different approaches tested on the Firefox dataset before and after handling negation, this shows how different approaches performed before and after handling negation on the same experimental settings.

Chapter 7

Source Code and Final Lexicons

This chapter provides an overview of the code files, data files, and static lexicon files used and generated in this study. This detailed explanation aims to assist the reader in running the code and generating the static lexicons. These files are available on the github repository, and the final lexicons are available on Zenodo, link provided below.
https://github.com/fatymaziz/DataAnalysis/tree/main/Experiments
https://zenodo.org/records/15591197

7.1 Data Files

The following table provides a brief description of each CSV file and the data contained within used in our study.

Table 7.1: Overview of the Dataset CSV Files

File Name	Description
bugs_Eclipse.csv	Contains the data of the Eclipse bug report used in our experimentation.
bugs_Firefox.csv	Contains the data of the Firefox bug report used in our experimentation.
bugs_Calendar.csv	Contains the data of the Calendar bug report for our experimentation.

7.2 Code Files

Following are the Python files that contain the code for learning, classification, and running experiments. The repository includes one file named helper.py which contains all methods for learning and classification, and fourteen additional Python files that act as controllers for running various experimental setups involving lexicon-based and machine learning methods.

7.3 Final Static Lexicon Files

The final static lexicons generated from our experiments are saved in JSON format and are also available in the static lexicon folder in the github repository with the following files.

Table 7.2: Descriptions and Functions of Python Code Files

File Name	Description
helper.py	Contains all methods for learning and classification across different approaches.
Experiment1.py	Contains code for experimental setup for dynamic lexicon generation and for the ML methods tailored to Eclipse dataset using threshold-based approach.
Experiment2.py	Contains code for experimental setup for dynamic lexicon generation and for the ML methods on the Firefox dataset using threshold-based approach.
Experiment3.py	Contains code for dynamic lexicon generation on the combined dataset of Firefox and Eclipse using threshold-based approach.
Experiment4.py	Contains code for dynamic lexicon generation on the combined dataset of Firefox and Eclipse with additional filters using threshold-based approach.
Experiment5.py	Contains code for static lexicon created on the Eclipse dataset and tested on the Firefox dataset using a threshold-based approach.
Experiment6.py	Contains code for static lexicon created on the Firefox dataset and tested on the Eclipse dataset using a threshold-based approach.
Experiment7.py	Contains code for static lexicon generation on the combined dataset of Firefox and Eclipse and ML methods using threshold-based approach.
Experiment8.py	Contains code for static lexicon generation on the combined dataset of Firefox and Eclipse with additional filters and for the ML methods using a threshold-based approach.
Experiment9.py	Contains code for static lexicon created on the Eclipse dataset and tested on the Firefox dataset using a linear SVM-based approach.
Experiment10.py	Contains code for static lexicon created on the Firefox dataset and tested on the Eclipse dataset using a linear SVM-based approach.
Experiment11.py	Contains code for classification of the Eclipse dataset using the Bing Liu Opinion Lexicon.
Experiment12.py	Contains code for classification of the Firefox dataset using the Bing Liu Opinion Lexicon.
Experiment13.py	Contains code for classification of the Eclipse dataset using VADER.
Experiment14.py	Contains code for classification of the Eclipse dataset using VADER.

Table 7.3: Overview of Lexicon JSON Files

File Name	Description
SEV_THR_LEX.json	Contains the final static lexicon after merging the generated lexicon from Firefox and Eclipse dataset using the threshold-based approach for classifying bug severity.
SEV_THR_LEX_c.json	Contains the static lexicon generated from the combined dataset of Eclipse and Firefox using the threshold-based approach.
SEV_THR_LEX_f.json	Contains the static lexicon generated from the combined dataset of Eclipse and Firefox but status with Normal bug filtered, using the threshold-based approach.
SEV_SVM_LEX.json	Contains the static lexicon generated using the SVM-based approach for classifying bug severity.

Chapter 8

Conclusions

This thesis has explored the potential of a lexicon-based approach for classifying bug severity, with the primary objective of determining whether such methodologies can serve as relevant baseline solutions for developing a robust general lexicon, since such a lexicon does not currently exist. To achieve this, static lexicons for software bug severity classification were created using two approaches and datasets from different sources.

Overall, this work has addressed the research questions in the following ways:

Can a lexicon-based approach be a competitive alternative to machine learning approaches for predicting the severity of software bugs?

The results of our study indicate that the performance of threshold-based, linear SVM-based approaches, and publicly available lexicons is influenced by the dataset used. The lexicons perform well in terms of the metric most relevant to our problem — the F1-score. However, this does not hold for the F1-mean, which suggests that machine learning methods perform better for the Non-Severe class.

The advantages of lexicon-based approaches — such as speed, effective classification performance, and transparency — typically make them viable alternatives to machine learning methods. Our results further suggest that lexicon-based approaches can indeed serve as practical solutions.

What is the efficiency of the lexicon-based approach in terms of time consumption compared to other methods?

The results indicate that the classification time of all lexicon-based approaches outperformed SVM, but did not match the efficiency of LogReg and MultiNB in the tested implementations. Notably, our lexicon learning and classification code was not time-optimized, so these observations are specific to the implementation used in this thesis.

How do our developed static lexicons perform against existing lexicons?

Our developed static lexicons performed well when compared with existing lexicons. However, performance was heavily influenced by the quality and domain of the datasets used during the lexicon creation process.

Publicly available lexicons were effective in classifying bug severity, and the results showed that even sentiment lexicons lacking software-specific vocabulary were surprisingly useful.

In some experiments, we also addressed negation handling. However, this did not lead to major improvements in performance. While slight improvements were observed in a few cases, in most experiments, the metrics were worse than before negation handling.

The scientific contributions of this work include the design and development of comprehensive and freely available general lexicons for software bug severity classification. By comparing their performance against existing lexicons and classic machine learning methods, this research provides valuable insights into the applicability and effectiveness of lexicon-based methods in this domain, ultimately contributing to more accurate and efficient bug severity classification practices.

In addition to lexicon-based and machine learning approaches, we also included a zero-shot evaluation of selected LLMs on the Firefox dataset. The results showed that llama3.3 and gemma3:27b outperformed our threshold-based lexicon approach for bug severity classification, while deepseek-r1:14b was less effective for the Severe class. These findings highlight the potential of modern LLMs for this task, even without additional training, and suggest that prompt-based LLM methods may offer a promising direction for future research.

In future work, we plan to improve the quality and coverage of our lexicon by using LLMs through prompt-based techniques. These models can be used to suggest related or synonymous terms, which could make the lexicon more complete and useful. This method would reduce the need for manual effort while still maintaining domain relevance. Additionally, once large LLMs become more affordable and practical to run, especially since current models require significant computing resources like GPUs, we aim to carry out a thorough comparison between our approach and LLM-generated output. This will allow us to better understand the strengths and limitations of each method in terms of accuracy, cost, and usability. We also plan to apply our improved approach to other domains where such lexicons are currently unavailable, which could help expand the benefits of this work beyond our original focus.

References

[1] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008, pp. 308–318.

[2] A. Kukkar, R. Mohana, A. Nayyar, J. Kim, B.-G. Kang, and N. Chilamkurti, "A novel deep-learning-based bug severity classification technique using convolutional neural networks and random forest with boosting," *Sensors*, vol. 19, no. 13, p. 2964, 2019.

[3] A. F. Otoom, S. Al-jdaeh, and M. Hammad, "Automated classification of software bug reports," in *proceedings of the 9th international conference on information communication and management*, 2019, pp. 17–21.

[4] J. Arokiam and J. S. Bradbury, "Automatically predicting bug severity early in the development process," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, 2020, pp. 17–20.

[5] T. Zhang, J. Chen, G. Yang, B. Lee, and X. Luo, "Towards more accurate severity prediction and fixer recommendation of software bugs," *Journal of Systems and Software*, vol. 117, pp. 166–184, 2016.

[6] A. Baarah, A. Aloqaily, Z. Salah, and E. Alshdaifat, "Sentiment-based machine learning and lexicon-based approaches for predicting the severity of bug reports," *Journal of Theoretical and Applied Information Technology*, vol. 99, no. 6, 2021.

[7] P. Li, *Jira Software Essentials: Plan, track, and release great applications with Jira Software*. Packt Publishing Ltd, 2018.

[8] K. K. Sabor, A. Hamou-Lhadj, A. Trabelsi, and J. Hassine, "Predicting bug report fields using stack traces and categorical attributes," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, 2019, pp. 224–233.

[9] L. A. F. Gomes, R. da Silva Torres, and M. L. Côrtes, "Bug report severity level prediction in open source software: A survey and research opportunities," *Information and software technology*, vol. 115, pp. 58–78, 2019.

[10] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *2010 7th IEEE working conference on mining software repositories (MSR 2010)*, IEEE, 2010, pp. 1–10.

[11] A.-H. Dao and C.-Z. Yang, "Severity prediction for bug reports using multi-aspect features: A deep learning approach," *Mathematics*, vol. 9, no. 14, p. 1644, 2021.

[12] A. Kaur and S. G. Jindal, "Text analytics based severity prediction of software bugs for apache projects," *International Journal of System Assurance Engineering and Management*, vol. 10, no. 4, pp. 765–782, 2019.

[13] W. Y. Ramay, Q. Umer, X. C. Yin, C. Zhu, and I. Illahi, "Deep neural network-based severity prediction of bug reports," *IEEE Access*, vol. 7, pp. 46 846–46 857, 2019.

[14] R. Agrawal and R. Goyal, "Developing bug severity prediction models using word2vec," *International Journal of Cognitive Computing in Engineering*, vol. 2, pp. 104–115, 2021.

[15] F. Calefato, F. Lanubile, F. Maiorano, and N. Novielli, "Sentiment polarity detection for software development," *Empirical Software Engineering*, vol. 23, pp. 1352–1382, 2017.

[16] G. Yang, S. Baek, J.-W. Lee, and B. Lee, "Analyzing emotion words to predict severity of software bugs: A case study of open source projects," in *Proceedings of the Symposium on Applied Computing*, 2017, pp. 1280–1287.

[17] T. Loughran and B. McDonald, "Textual analysis in accounting and finance: A survey," *Journal of Accounting Research*, vol. 54, no. 4, pp. 1187–1230, 2016.

[18] M. R. Islam and M. F. Zibran, "Sentistrength-se: Exploiting domain specificity for improved sentiment analysis in software engineering text," *Journal of Systems and Software*, vol. 145, pp. 125–146, 2018, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2018.08.030.

[19] G. Sharma, S. Sharma, and S. Gujral, "A novel way of assessing software bug severity using dictionary of critical terms," *Procedia Computer Science*, vol. 70, pp. 632–639, 2015.

[20] T. Olaleye, O. Arogundade, S. Misra, A. Abayomi-Alli, and U. Kose, "Predictive analytics and software defect severity: A systematic review and future directions," *Scientific Programming*, vol. 2023, no. 1, p. 6 221 388, 2023.

[21] E. Mashhadi, H. Ahmadvand, and H. Hemmati, "Method-level bug severity prediction using source code metrics and llms," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2023, pp. 635–646.

[22] S. Guo, R. Chen, H. Li, T. Zhang, and Y. Liu, "Identify severity bug report with distribution imbalance by cr-smote and elm," *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 02, pp. 139–175, 2019.

[23] B. Liu, *Sentiment analysis: Mining opinions, sentiments, and emotions*. Cambridge university press, 2020.

[24] G. A. Miller, "Wordnet: A lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[25] M. Topaz, L. Murga, O. Bar-Bachar, M. McDonald, and K. Bowles, "Nimbleminer: An open-source nursing-sensitive natural language processing system based on word embedding," *CIN: Computers, Informatics, Nursing*, vol. 37, no. 11, pp. 583–590, 2019.

[26] K. Takeoka, K. Akimoto, and M. Oyamada, "Low-resource taxonomy enrichment with pretrained language models," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 2747–2758.

[27] S. M. Islam, X. L. Dong, and G. De Melo, "Domain-specific sentiment lexicons induced from labeled documents," in *Proceedings of the 28th International Conference on Computational Linguistics*, 2020, pp. 6576–6587.

[28] M. Hu and B. Liu, "Mining and summarizing customer reviews," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004, pp. 168–177.

[29] S.-M. Kim and E. Hovy, "Determining the sentiment of opinions," in *Coling 2004: Proceedings of the 20th international conference on computational linguistics*, 2004, pp. 1367–1373.

[30] C. Hutto and E. Gilbert, "Vader: A parsimonious rule-based model for sentiment analysis of social media text," in *Proceedings of the international AAAI conference on web and social media*, vol. 8, 2014, pp. 216–225.

[31] R. Bibyan, S. Anand, and A. Jaiswal, "Assessing the severity of software bug using neural network," in *Strategic System Assurance and Business Analytics*, P. K. Kapur, O. Singh, S. K. Khatri, and A. K. Verma, Eds. Singapore: Springer Singapore, 2020, pp. 491–502, ISBN: 978-981-15-3647-2. DOI: 10.1007/978-981-15-3647-2_35. [Online]. Available: https://doi.org/10.1007/978-981-15-3647-2_35.

[32] S. K. Hamouri, R. A. Shatnawi, O. AlZoubi, A. Migdady, and M. B. Yassein, "Predicting bug severity using machine learning and ensemble learning techniques," in *2023 14th International Conference on Information and Communication Systems (ICICS)*, 2023, pp. 1–6. DOI: 10.1109/ICICS60529.2023.10330494.

Bibliography

Publications Related to the Thesis

Conference Paper

F. Aziz and M. Žnidaršič, "Potential of lexicon-based bug severity classification," in *14th International Conference on Information Technologies and Information Society (ITIS 2023)*, Ljubljana, 2023, pp. 21–28.

F. Aziz and M. Žnidaršič, "Classfication of bug severity with lexicon approach," in *15th International Conference on Information Technologies and Information Society (ITIS 2024)*, Ljubljana, 2024, pp. 196–205.

Biography

Fatima Aziz was born in Hunza Valley, Pakistan. She completed her primary and secondary education in her hometown. In 2015, she obtained her bachelor's degree in Computer Science from COMSATS Institute of Information Technology, Islamabad, Pakistan. After graduation, she began her career in the software industry, working as a Java developer and software quality assurance engineer with local and international companies. In 2021, she enrolled in a master's degree program at the Jožef Stefan International Postgraduate School as a master's student under the supervision of Assist. Prof. Dr. Martin Žnidaršič. During this time, she worked on developing a lexicon-based software bug severity classification method. She presented her work at the Information Technologies and Information Society (ITIS) conferences in 2023 and 2024 in Ljubljana. In November 2024, she was employed at the Jožef Stefan Institute and began working at the Department of Knowledge Technologies, primarily in the scope of the project DIGITOP (Digital transformation of robot-supported factories of the future).