Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Software Engineering and Technology Research Group

Tracing Evolutionary Changes of APIs

Master Thesis

Kristiyan Michaylov

Supervisor:

Dr.-Ing. Jacob Krüger

Committee Members:

Prof. Dr. Michel Chaudron
Dr. Ir. Tim Willemse

1.0

Eindhoven, June 2025

Abstract

In modern software development, APIs are essential tools that enable developers to utilise the capabilities provided by software libraries efficiently. As any software artifact, APIs evolve, undergoing changes in functionality and architecture. Therefore, understanding their evolution is essential for creating robust and adaptable systems. The current study emphasises and addresses the importance of understanding the underlying causes of API modifications. Particularly, their impact on software functionality and architecture. To explore this, we utilise Java APIs from Maven, including JUnit, Project Lombok, Log4J and Apache Commons IO. Our study involves an in-depth analysis of the aforementioned codebases and corresponding datasets featuring release log messages for selected API versions. We implement machine learning based techniques for categorising the release log messages and utilise empirical methods for analysing the functionality and architecture of the codebases. Our focus is on class-level architecture. Our findings show that machine learning models can classify release log messages with above 70% accuracy in the majority of cases. Moreover, detailed and descriptive release message logs significantly enhance classification accuracy, while brief messages hinder it. Concerning the architecture and functionality, the results suggest that codebases evolve predominantly through adding new classes and layers of abstraction. Design patterns, such as Visitor and Strategy, are utilised and extended throughout evolution, indicating a structured approach related to the evolution process. Overall, this research contributes to the understanding of API evolution. Particularly, it highlights the reasons for changes in APIs and their impact on architectural and functional aspects. These insights can help software professionals manage the effects of such changes.

Preface

This thesis concludes my Master's endeavours in Computer Science and Engineering at the Eindhoven University of Technology. In this research, part of my focus was on topics such as natural language processing and machine learning, areas that were initially outside of my core strengths and competencies. At the same time, I explored software architecture, which has always fascinated me. Working on this project challenged me to step outside my comfort zone and deal with difficult situations and choices. This experience taught me that with the right mindset and persistent effort, challenges can be overcome.

First and foremost, I would like to thank my supervisor, Jacob Krüger. He provided me with an opportunity to work on this interesting research topic. His continuous support, insightful guidance, and readiness to answer my questions during our meetings were invaluable. I would also like to express my sincere gratitude to Michel Chaudron and Tim Willemse for agreeing to act as committee members. Their input during the preparation phase presentation was essential in shaping the final thesis variant. Both provided perspectives that I had not previously considered, which proved to be crucial for my research paper.

I would also like to extend my sincere thanks to Alexandra Nikolova. Without her foundational work on the topic, completing the current study would have been significantly more challenging. Last but not least, I express my appreciation towards my family and friends. Their support was invaluable, encouraging me to persevere through challenging moments and maintain my focus.

This is the final step of my university education. Over the past five years in the Netherlands, I have acquired valuable knowledge and experiences from the University of Twente and Eindhoven University of Technology that will be foundational in shaping my future career. In conclusion, learning is a lifelong process, and the exchange of knowledge is an ongoing, ever-evolving phenomenon.

Acronyms

API Application Programming Interface.

BC Breaking Changes.

DL Deep Learning.

FN False Negative.

FP False Positive.

MCR Maven Central Repository.

ML Machine Learning.

NBC Non-Breaking Changes.

RFC Random Forest Classifier.

SC Stacking Classifier.

sklearn scikit-learn.

SVC Support Vector Classifier.

SVM Support Vector Machine.

TN True Negative.

TP True Positive.

VC Voting Classifier.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems have become progressively more complex over the last three decades. Nowadays, developing software products requires integration and communication with various components that should function as a coherent system. However, connecting and building software components can be challenging, particularly when the majority of the work must be performed from scratch. To circumvent this inconvenience, software engineers utilise reusable software components, namely **software libraries**. To access these libraries and utilise their functionality, **Application Programming Interfaces (APIs)** are employed. APIs play a crucial role in software development, providing widely adopted mechanisms for enabling system integration and functionality. Their main tasks are facilitating communication and data exchange between software components to reduce elaboration effort and encourage software re-usability [3]. As a result, developers can reuse available functionalities and avoid unnecessary implementations from scratch [4]. Such actions are crucial for enterprises, as they help save time and financial resources. Over the past 20 years, extensive explorations have been conducted on APIs. Primarily, researchers have been interested in the technical aspect, particularly their design and utilisation. Additionally, another highly studied topic has been the evolution of APIs [3].

Software evolution has been acknowledged as a challenging and fundamental subject in computer science since the 1970s. Back then, Lehman formulated a framework for understanding it in the context of software engineering [5]. The term evolution refers to the dynamic changes in software systems to accommodate necessary maintenance and enhancements during their lifetime [6]. Since APIs are digital artifacts, they are also susceptible to evolution and Lehman's laws [3, 5]. Consequently, APIs undergo continuous modification and refinement over the course of their lifecycle [3].

The growing interest on API evolution has resulted in a 2021 study by Lamothe et al. [3]. It presents a significant literature review on the topic, covering the period from 1994 to 2018. Overall, the authors find that 70.4% of the research presented in the examined papers focuses on Java APIs. Furthermore, these studies often focus on API usability and maintainability. Additionally, the authors suggest that future research should be performed on automated traceability between APIs. A specific topic that has not been thoroughly explored is the causes behind API changes. Moreover, little is understood about how such changes impact the externalisation of their architecture and functionalities. This information is crucial since APIs are widely adopted. Thus, understanding the patterns of evolution can help to identify common systems or architectural changes that require API adaptations. Our research seeks to address this gap.

Java has been one of the most popular and widely utilised programming languages for almost three decades [7, 8]. As a result, Java APIs and their ecosystem are versatile and rich in functionality. Multiple researchers [2, 9, 10, 11] have focused on studying Java APIs. A suitable platform for this purpose is the *Maven Repository*, which hosts a vast collection of Java APIs and software libraries [12]. The repository features notable software libraries such as JUnit, SLF4J, Apache Commons, Mockito, Lombok, and more [12, 13]. They serve as key landmarks in the Java ecosystem, relied upon by hundreds of thousands of users. Thanks to its extensive contributions and

usage, the Maven Repository has been established as an excellent and intriguing place to study the evolution of APIs.

As mentioned in the previous paragraphs, APIs are susceptible to evolving. The evolution of a software artifact often necessitates changes to its functionality and underlying architecture. To elaborate on the concept of software architecture, we define it as the high-level design and organisation of a software system. For instance, a class diagram that illustrates a subset of the relationships and interactions between classes can be viewed as a high-level representation of a system's design. It allows one to infer the structure of a software system, but not its specific functionalities. High-level system architecture can be organised and structured utilising various architectural styles. Common examples include *Pipes and Filters*, *Layered Architecture*, *Client-Server*, and *Microservices* [14]. During the development or maintenance of software libraries accessed via APIs, sometimes it is necessary to change part of the system's architecture. Such alternation can cause changes in the functionality and architecture of the API. This signifies another interesting direction that is not extensively explored in research.

1.1 Problem Statement

In the previous section, we explained some important points and topics regarding APIs, such as evolution, architecture & functionalities and language-specific APIs. Nevertheless, researchers have studied various other topics. Those include investigations on breaking changes, the role of refactoring on APIs, semantic versioning and more [3, 11, 15, 16]. A notable gap in research is the understanding of the underlying causes of API changes. Specifically, there has been limited investigation into how these changes impact the externalisation of API architecture and functionality. This is a significant and relevant area of research, as it aids in understanding whether the architectural changes require adaptations within the API. Such insights are crucial, given the widespread usage and reliance on APIs. Moreover, such a suggestion has been proposed by some studies as a potential direction for future research [3]. With the current investigation, we aim to explore this gap. As a result, we formulate the following main goal:

> ***Main Goal**: Analyse the reasons for changes in APIs and the impact of such alterations on the functionalities and the architecture of the APIs.*

1.2 Research Questions

Given the problem context and main goal we will aim to answer the following research questions:

> RQ_1 : *To what extent can an automated machine learning technique analyse and categorise the causes of changes in Java APIs?*

By addressing this, we aim to investigate the extent to which the reasons behind API changes can be analysed automatically. This approach can help scale categorisation efforts in future work and save time. We hypothesise that a correct categorisation regarding the causes of changes in Java APIs can be achieved with above 70% accuracy.

> RQ_2 : *How do changes impact the architecture and functionalities of APIs?*
>
> > $RQ_{2.1}$: *Are changes impacting the architecture and functionality represented in the release log?*
> >
> > $RQ_{2.2}$: *How does the architecture of an API alter during evolution?*

After categorising the causes of changes in Java APIs, the next step is to evaluate the architecture. Answering RQ_2, along with its sub-questions, supports addressing the second part of the main objective. Therefore, empirically observing patterns in which changes impact the architecture and functionality of APIs. Furthermore, this includes examining whether such alterations are reflected in the release log messages of the corresponding versions.

1.3 Thesis Outline

The rest of the thesis is structured as follows:

- In chapter 2, there is important background information regarding APIs.
- In chapter 3, we share the relevant literature review on topics related to APIs and evolution.
- In chapter 4, the proposed methodology is outlined, alongside argumentation of what and why is considered.
- In chapter 5, we outline the obtained results for both research questions and discuss their implications.
- In chapter 6, we list the threats to validity connected with the research approach.
- Finally, in chapter 7, we recapitulate the contents of the thesis and propose directions for follow-up studies.

Chapter 2

Background

In this chapter, we aim to provide the reader with foundational concepts necessary to understand this research. Those include domain-specific terminology and theoretical matters utilised throughout the following sections. Therefore, we explain background knowledge related to breaking changes, software versioning, software evolution, software metrics and software architecture, alongside important machine learning (ML) techniques, concepts and evaluation metrics. This is essential for understanding the experimental work conducted in this research and the related work for APIs.

2.1 Breaking Changes

Generally, in research, changes in APIs are categorised as either breaking changes (BC) or non-breaking changes (NBC). BC are defined as incompatible with previous versions. Client projects that rely on a specific API entity impacted by a BC may encounter problems when upgrading to a newer version of the API [9, 11]. On the other hand, NBC are backwards compatible. This means that such changes can be integrated into the application without affecting its behaviour [11].

2.2 Software Versioning

In software, it is crucial to provide a mechanism for tracking changes. Versioning in APIs is a source which can be utilised for understanding how changes impact them [3, 17]. Various versioning schemes exist, such as Semantic Versioning (SemVer), Calendar Versioning (CalVer), Zero Versioning (ZerVer) [18]. In this paper, we focus our attention on SemVer. It is a scheme that defines a set of rules and requirements which determines how version numbers can be assigned and altered [19]. This is performed using the format *MAJOR.MINOR.PATCH*, where the *MAJOR* number is incremented when you perform incompatible API changes. The *MINOR* is when backwards-compatible functionality is added, and the *PATCH* is when backwards-compatible bug fixes are applied[19]. Figure 2.1 summarises the semantic versioning format. One of the motivations for such standardisation is the phenomenon called *Dependency Hell*. This occurs when some software is entangled with specific versions of other software, preventing the developers from moving forward [19]. Overall, the topic of semantic versioning is essential and strongly connected with API evolution.

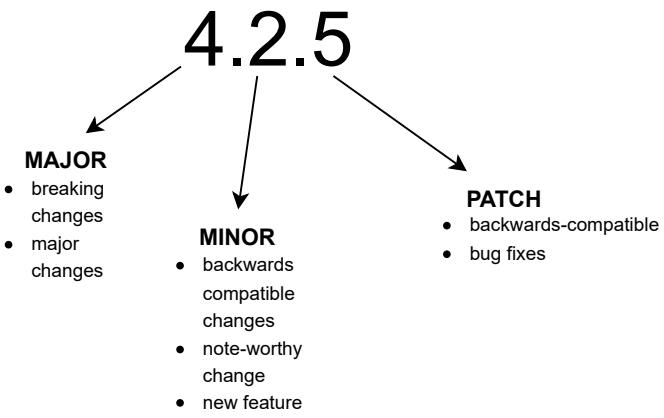


Figure 2.1: Visualisation for the semantic versioning format

2.3 Software Evolution

During its lifecycle, software systems require continuous adaptation and improvement to facilitate future development and maintainability. Refactoring is an essential concept in API evolution and software as a whole. Martin Fowler, in his book on refactoring [20] defines it as restructuring software without altering its observable behaviour. Some famous refactoring examples include **Extract Method**, **Move Field**, **Replace Conditional with Polymorphism**, and more [20]. The general purpose of refactoring is to improve the codebase via a series of changes. As a result, the codebase should be easier to maintain, less coupled and error-prone. With respect to APIs, refactorings encompass a crucial role in the evolution. For instance, they may serve as either a direct or indirect reason for an API version upgrade.

2.4 Software Architecture

Comprehending the structure of software systems and the interactions between their components is a critical task for software developers. This information aids in implementing new features, understanding connections between components, etc. Software architecture is a concept which provides this crucial information. The ISO/IEC/IEEE 42010 standard defines it as "Fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution" [21]. This definition infers that software architecture is a broad concept which includes various sub-parts such as views, viewpoints, and styles [14]. Architectural styles are utilised to show the structural and organisational schema for software systems [14]. Various architectural styles exist, such as Client-Server, Pipes and Filters, Peer-to-Peer, among others. Overall, understanding the concept of software architecture as a whole is essential for the current research and has an integral role throughout the study.

2.5 Software Metrics

Software metrics are properties that enable the quantifiable measurement of software characteristics [22]. They can be utilised to obtain quantifiable insights regarding software development, such as code quality, complexity, maintainability, performance, and reliability. Nonetheless, metrics cannot be directly applied to infer properties of software, such as code quality or complexity,

on their own. For instance, a high metric value may not always indicate poor quality or high complexity without proper context. Therefore, it is important to attach meaning and context to them [23]. Specific software metrics include lines of Java code, number of packages, number of classes, and number of methods. The first metric is computed by counting the number of lines that possess Java code, excluding blank space. The remaining ones are obtained by counting the number of occurrences of the corresponding code element. Overall, the information about metrics is valuable for gaining an overview of the codebase.

2.6 Machine Learning Techniques

Machine learning allows computer systems to leverage algorithms and statistical models for data analysis and pattern detection. Training a model on a given dataset enables the system to make predictions on previously unseen data. Several types of ML exist, including *Supervised Machine Learning*, *Unsupervised Machine Learning*, *Semi-Supervised Learning* and *Reinforcement Learning*. Figure 2.2 depicts the type of MLs, alongside their subtypes. In the context of our study, we emphasise the *Supervised ML*. This is a category that involves training a function to map inputs to outputs based on a set of input-output examples [24]. As displayed in Figure 2.2, this category has two subtypes of ML techniques, namely *classification* and *regression*. In classification, a function is utilised to separate data into respective classes. As shown in Figure 2.3, the goal is to accurately assign each data point to one of the predefined categories. In contrast, regression aims to model the relationship between variables by fitting a function to the data. In the current study context, *classification* is the selected technique [24]. Figure 2.3 illustrates how each of the two subtypes works. Overall, as the goal is to categorise release log changes, a classification approach proves to be the most suitable for our experiment.

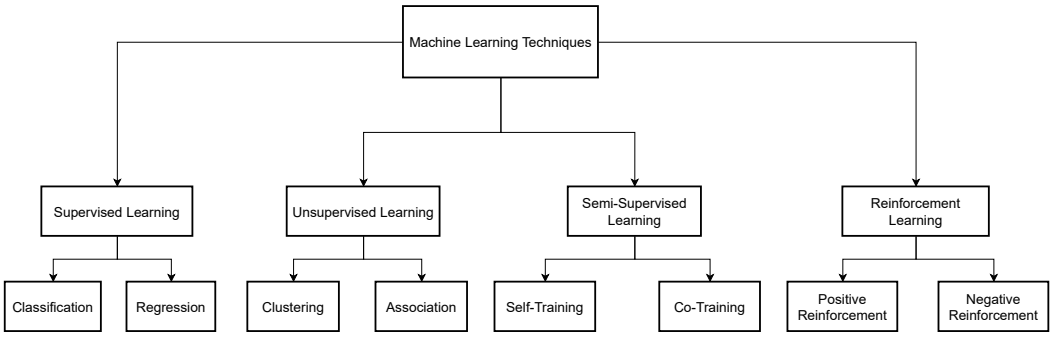
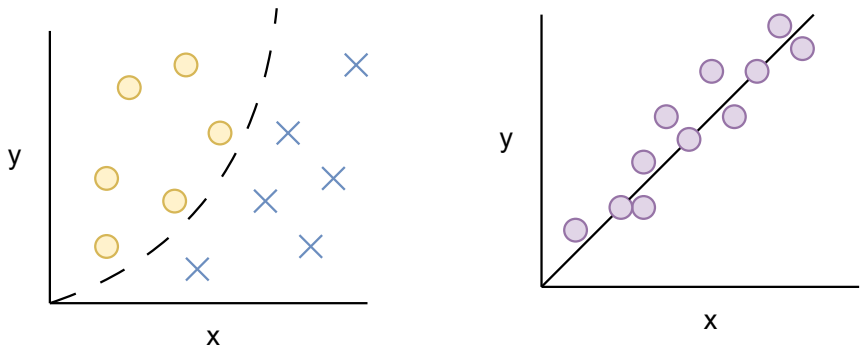


Figure 2.2: General categorisation of machine learning techniques



(a) Classification (b) Regression

Figure 2.3: Comparison between classification and regression approaches

2.7 Natural Language Processing

Natural Language Processing (NLP) is a branch of artificial intelligence (AI) that incorporates elements of linguistics to enable machines to understand, interpret, and manipulate human language [25]. NLP has various practical usages, with prominent examples including *text categorisation*, *sentiment analysis*, which classifies the emotional intent of text and *speech recognition*. The techniques included in this study are primarily related to text categorisation, as this aligns directly with the study's main goal and the corresponding *RQs*. To successfully categorise a text, several techniques have to be employed. The intent is to prepare the raw text in a form suitable for being an input of a ML model, which can then effectively categorise the input with a high degree of accuracy. Such techniques include stop-words removal, stemming and lemmatization, tokenisation and TF-IDF. The specific in-depth information regarding the NLP techniques usage is provided in the Methodology section. This is motivated by the need to present all steps involved in categorising release log messages. Thus, enhancing the reader's understanding through a systematic presentation of the information.

2.8 Machine Learning Evaluation Metrics

Evaluation metrics offer a means to assess the performance of machine learning models. Most utilised metrics include accuracy, precision, recall, and F1-score. However, computing these requires an understanding of the foundational components that build them. These foundational concepts include the notion of True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN). Comprehending these terms is essential for accurately interpreting the performance metrics and their significance in evaluating model effectiveness. To aid this process, Table 2.1 summarises the meaning of each characteristic.

Table 2.1: Confusion Matrix Terminology

	Predicted Positive	**Predicted Negative**
Actual Positive	**True Positive (TP)** Correctly predicted as positive	**False Negative (FN)** Incorrectly predicted as negative
Actual Negative	**False Positive (FP)** Incorrectly predicted as positive	**True Negative (TN)** Correctly predicted as negative

We now focus on the evaluation metrics. Equation 2.2 lists the formulas of all of them, starting with accuracy. This is a simple metric which establishes the extent to which a model correctly classifies the data. Although accuracy is an important metric, it cannot provide in-depth value alone. Particularly, it is susceptible to imbalanced classes of data. For instance, when one class dominates, a model may appear highly accurate due to favouring the majority class, while neglecting and low accuracy on the minority class. To mitigate this, we explored other metrics, such as recall, precision and F1-score. The recall addresses the number of actual positive data points that are predicted to be positive. Precision establishes the proportion of TP among the positive predictions. Finally, the F1-score is simply the harmonic mean of precision and recall. These three metrics present different angles for interpreting the data and, combined with accuracy, assist in balancing trade-offs in model performance.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \qquad \text{Precision} = \frac{TP}{TP + FP} \tag{2.1}$$

$$\text{Recall} = \frac{TP}{TP + FN} \qquad F1\text{-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.2}$$

Chapter 3

Related Work

In this chapter, we focus on relevant related work, connected to the topic of APIs. As briefly explained in the introduction, diverse research has been performed on APIs. Here, we concentrate on three crucial themes for the current study. One is *API Evolution*, where papers focus on how evolution is happening in APIs. Afterwards, we investigate the performed research on *Semantic Versioning* to check its impact and usage in APIs. Next, we explore *Language-Specific API Evolution* to study how different software libraries evolve. Finally, we review literature on *Architecture and APIs* to explore how architectural elements and functionalities relate to API changes, and how various tools are employed in this context. All consulted papers are placed in Table 3.1

Table 3.1: Summary of discussion topics addressed in the reviewed literature

Topic	**Papers**
API Evolution	[2], [3], [9], [10], [15], [26]
Semantic Versioning	[11], [16], [17], [27]
Language-Specific API Evolution	[2], [15], [28], [29], [30]
Software Architecture	[9], [31], [32]

3.1 API Evolution

In the introduction section, we explained that API Evolution refers to the alterations of APIs in the course of their lifetime [3]. This concept is crucial since APIs are widespread and utilised in various domains.

A study by Lamothe et al. [3] is a notable paper investigating this topic. This systematic literature review analyses publications on API evolution from 1994 to 2018. Overall, the findings indicate an increase in the number of studies addressing API evolution over the past 25 years. This underscores the rising importance of research on this theme. Another important point mentioned is that research is mainly concentrated on Java APIs, with 70.4% of papers exclusively using them. This denotes a bias towards the particular programming language and a possibility for replication between studies [3]. Additionally, it would not always be feasible to generalise the results obtained from Java APIs for other languages. This is another problem shared by the authors. As explained in the introduction, the research on Java APIs often focuses on practical challenges. Therefore, topics such as usability and maintainability, including breaking changes, integration problems, and change impact, are more emphasised, compared to others [3]. The paper outlines the importance of future research into the potential for automatic identification and analysis of the reasons behind API changes.

In a paper by Koci et al. [26], the authors attempt to provide a comprehensive understanding of API evolution. This is realised by identifying and classifying the changes that occur to APIs

throughout their life cycle. Furthermore, the study investigates API evolution through the lens of actors, such as producers and consumers. The authors argue that from the consumer point of view, changes can be classified as backwards-compatible and breaking changes and emphasise that producers should use artifacts to document such changes. Example artifacts are release notes, API documentation, issue tracker, and versioning system. Due to such compatibility problems, users do not upgrade to newer API versions unless faced with critical bugs or the need for new features. This indicates an inconsistency between API evolution and consumer adaptation.

A seminal work by Dig and Johnson [9] investigates principles by which APIs evolve. Important topics addressed in API evolution are breaking and non-breaking changes. Based on five Java systems, the study has found that in a substantial majority of the cases, the breaking changes are refactorings. The authors suggest that more attention should be focused on better understanding the types of changes that occur in APIs. Additionally, tools should be created to facilitate and automate the process of upgrading applications to new API versions.

An empirical investigation by Kim et. al. [10] concentrates on the API-level refactorings in software evolution. Based on case studies of three Java-based systems, the study suggests that the number of bug fixes increases after refactorings. However, the time needed to fix those bugs decreases after refactoring. This is interesting since the main task of refactoring is to change the structural aspects of a code fragment without altering the behaviour. As a result, reduce the technical debt [10].

3.2 Semantic Versioning

As emphasised in the background chapter, versioning is essential in understanding and tracking API evolution. In the current section, we focus on papers that investigate the topic of semantic versioning and are connected with API evolution.

Raemaekers et al. [16] focus on examining 100,000 open-source libraries in Maven Central. They analysed how effectively the semantic versioning scheme, as defined by the library developers, showcases breaking changes to library users in a given release. The results from the study indicate that approximately one-third of all releases introduce at least one breaking change. This produces compilation errors in client libraries, which need to be fixed before upgrading. The authors recommend that the principles of semantic versioning should be more widely adopted and consistently applied by developers.

A more recent paper by Ochoa et al. [11] performs an external and differentiated replication study of the one by Raemaekers et al. [16]. Their research aims to understand which kind of Breaking Changes happen in libraries on the Maven Central Repository (MCR) and their impact. It has been found that semantic versioning is being applied in most cases (83.4% on MCR) and that the respective principles of introducing Breaking Changes are followed. Furthermore, in the period 2005 to 2018, the usage of semantic versioning has increased. As a result, the number of BCs has gone from 67.5% at the beginning period to 16.0 % in 2018, showing a positive influence.

As discussed in the previous paragraph, semantic versioning has been increasing its popularity. However, not all developers utilising it adhere to the guidelines. A recent study by Serbout and Pautasso [17] performed an empirical research on 3,075 Web APIs. The authors found that 2,282 APIs have released BC as minor or patch updates. Additionally, some Web APIs do not use semantic versioning and provide limited information to users regarding the impacts of the new version [27]. In such cases, semantic versioning has been suggested as a possible improvement and separate releases for bug fixes and new features [27].

3.3 Language-Specific API Evolution

Our current research is influenced by the Master's thesis paper by Alexandra Nikolova [2]. In her thesis, she focuses on the reasons for the evolution of Java APIs by examining three aspects. What are the causes of Java library evolution, what are the consequences of such evolution, and how ex-

actly do Java libraries evolve? The study categorises the causes of API changes into 14 categories. The most prominent being *Bug fix*, *Add support for a new feature*, *Documentation*, *Code redesign*, and *Maintainability improvement*. This information was obtained based on the investigation of eight libraries (APIs) from the Maven repository. Concerning the consequences of evolution, the study suggests that changes in libraries have positive and negative ones [2]. The positive includes improved functionality and usability through bug fixes and feature additions. Such changes may also negatively impact developers, e.g., the need for increased maintenance, documentation, and testing efforts. Furthermore, alterations such as code redesign and functionality deprecation can require significant adaptations in user applications. Moving to the evolution of Java libraries, the author has identified three important principles. With respect to documentation, most libraries document their evolution in their source repositories or websites. This is achieved via similar patterns observed among libraries under the same license, such as Apache. Different classification patterns are utilised for libraries with different licenses. Regarding the release, libraries vary in release frequencies. Most releases typically occur early in the life cycle and constitute a decline in later periods. Ultimately, regarding the changes per release, it was determined that MINOR and PATCH releases dominate over MAJOR ones. Furthermore, some libraries decide to push many MINOR releases, while others prefer the PATCH ones. Generally, across all release types, bug fixes and new feature additions are the most frequent changes. Bug fixes are especially prevalent in PATCH releases. As a final remark, the paper proposes the exploration of libraries from different categories and repositories. This would enhance the diversity of the results obtained.

Brito et al. [15] performed a study using a fire-house interview method on 400 popular Java APIs and libraries. In particular, they investigated how developers introduce breaking changes. The study found that most of them are introduced after refactorings, particularly on methods. This information is intact, and we have cited previous studies. Furthermore, the reasons for breaking APIs can mainly be organised into four themes, namely, *Addition of new features*, *API simplification*, *Maintenance* and *Bug fixing*. As a future direction, the authors suggest investigating other programming languages and the impact of breaking changes there.

Outside of Java, researchers have conducted limited studies on API evolution [28, 29, 30]. Even so, there are papers which depict interesting results. Zhang et al. [30] have performed a systematic exploration study on the evolution of Python Framework APIs. Based on an analysis of 288 releases from six popular Python frameworks, they observed 14 evolution patterns in them. Those include Method Removal, Class Removal, etc. Compared to Java, Python API evolution has introduced more breaking changes.

3.4 Architecture and APIs

Software architecture has been a well-studied topic, especially considering the high-level concepts. Books and articles [33, 34, 14, 31, 9] have been composed on the topic of software architecture. The investigation concerning the impact of API evolutionary changes on software design and architecture has been scarce. Moreover, it could be beneficial to investigate the alternation of software architecture, utilising existing tools. This way, it could be argued how and why the design of software changes. Thus, we share research in this section that can help explore the gap.

A paper by Rukmono et al. [31] introduces a novel visualisation technique aimed at enhancing the understanding of software architecture through a layered approach. It outlines the concept of layered architecture, which typically consists of four main layers, presentation, service, domain, and data source. Each of them has specific responsibilities. BubbleTea addresses two main purposes. First, it maps software components to architectural layers. Second, it emphasises the contents of software components based on their alignment with the typical functionality of the four architectural layers. The authors address the prevalent issue of architectural layer violations, where actual implementations deviate from intended structures, highlighting the need for effective visualisation tools. BubbleTea examines the alignment of software components with their layers' stereotypical responsibilities, providing deeper insights into architectural integrity.

Rukmono further upgrades BubbleTea by introducing a newer version of it, namely bubbleTea

2.0. The latter one is an improvement, utilising a deductive software architecture recovery technique [32]. This technique works by beginning with an abstract reference architecture, defined by key components and indicators. Using chain-of-thought reasoning, GPT-4 analyses source code to match features to these indicators, classifying behaviour deductively. These classifications are then aggregated into a full system architecture, with GPT-4 offering clear explanations of its reasoning. In the end, the method is especially useful for increasing accuracy, transparency, and system understanding when documentation is insufficient or lacking.

The study by Dig and Johnson [9] investigates the role of API changes in software architecture. There, the authors argue that API evolution significantly impacts architecture and functionality through structural and behavioural alterations. Structural changes, comprising over 80% breaking API changes, primarily involve refactorings. Those include renaming or moving methods and fields, altering method signatures, and introducing new interfaces. The purpose of such changes is to improve maintainability and reusability while preserving internal behaviour. Nevertheless, it is found that this often leads to breaking compatibility with client applications. Conversely, changes in behaviour, such as updates to API contracts, may affect how the application operates. These changes, though often aimed at enhancing functionality, pose challenges for maintaining backwards compatibility. Furthermore, they require thorough testing and clear documentation to mitigate their impact on dependent systems.

3.5 Conclusions

The literature shows APIs have been a highly researched theme in the last 27 years. Research interests span topics such as evolution in APIs [3, 26, 9], semantic versioning [16, 11], and studies specifically on Java libraries mainly from the Maven repository [2, 15, 9] and papers on architecture and functionalities regarding APIs [9, 31]. In future directions, researchers suggest investigating subjects such as evolution APIs from Java and automatic means to analyse the reasons for API changes. Thus, the current research aims to explore these gaps while further concentrating on the functional and architectural implications of such changes.

Chapter 4

Methodology

To address the main research goal and its associated questions, a combination of methodological approaches was employed. In this chapter, the aim is to explain the proposed methodology alongside intermediate steps. Furthermore, we discuss the potential advantages and disadvantages of the selected approach. To ensure better clarity, the chapter is divided into several sections. The first section outlines the identified general approach for the corresponding research questions. In the second section, we reveal the data collection, extraction and categorisation steps. Afterwards, the entire training setup for the machine learning model is explained, including techniques, parameters, and evaluation metrics. The final two sections outline the phases of architectural analysis and the tools chosen to address both research questions. For illustrating the study's methodology, Figure 4.1 provides a comprehensive overview of the entire process.

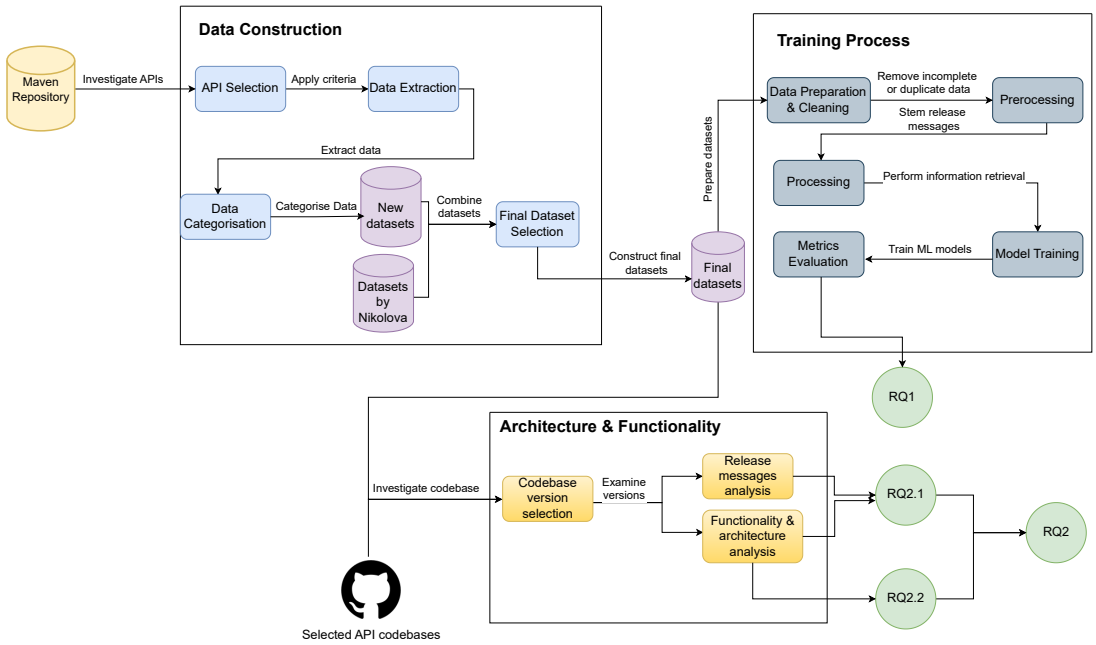


Figure 4.1: Diagram capturing the whole Methodology process

4.1 Approach

This section outlines the planned methodology approach for addressing all research questions. The discussion begins with a recapitulation of them, accompanied by the corresponding approach for each.

> RQ_1 : To what extent can an automated machine learning technique analyse and categorise the causes of changes in Java APIs?

The initial point of the research was concentrated on the development of an automatic machine-learning technique that can analyse changes and evolutions in APIs. To do this, we utilised information from release notes, since they are easier to analyse, shorter and provide higher-level information. For this study, we employed a portion of the dataset compiled by Nikolova [2], constructed from Maven libraries. Moreover, two software libraries constructed following the format by Nikolova were added. To facilitate examination of the dataset structures, the data are available as CSV files at the following link. On the datasets, experiments were performed with machine learning techniques, including *Natural Language Processing (NLP)* and *Classification* to categorise the changes. The objective was to categorise the change based on the changelog text. The preliminary goal for the accuracy of such a model was to achieve at least **70%** per dataset. Determining this value as a threshold should provide a degree of certainty that the model works correctly in most cases. Nevertheless, we acknowledged that evaluating singly by accuracy can feature caveats. To mitigate this, we considered the utilisation of other metrics. Those included precision, recall and F1-score that would have indicated if the model produced a high percentage (e.g. above 50%) of false positives or negatives.

> RQ_2 : *How do changes impact the architecture and functionalities of APIs?*
>
> > $RQ_{2.1}$: *Are changes impacting the architecture and functionality represented in the release log?*
> >
> > $RQ_{2.2}$: *How does the architecture of an API alter during evolution?*

Once the automatic technique was developed and tested, we attempted to identify how changes influence the architecture and functionalities of APIs. The study focused on a subset of the APIs from Maven Central that were selected for RQ_1. For this purpose, architectural visualisation tools, generating class diagrams and dependency graphs, were applied to selected API versions. Additionally, descriptive metrics—such as the number of classes and methods per version—were obtained to understand the size and other high-level attributes of the system. This information, along with an analysis of the release log messages for the selected versions, provides a way to evaluate how functionalities and architecture change over time, and whether those modifications are accurately reflected in the release logs.

4.1.1 Research Design

In the previous section, we outlined the proposed approaches for addressing each research question. As is evident, answering these questions requires a combination of techniques. Therefore, we had to implement a technique that analyses release messages of APIs. Additionally, we derived our results primarily from observation or experimentation combined with some theoretical analysis. As a result, our work could be classified as a case-study research featuring empirical evaluation. Furthermore, it employs a mixed-methods approach.

4.2 Data Collection

In this section, we outline the steps taken to collect data for the datasets. It was crucial to construct suitable ones in order to analyse the possible automatic techniques. First, we explain

the criteria for selecting APIs. Based on it, we share the Java APIs which were selected and indicate whether they were newly created by us or already available from the work by Nikolova [2]. Finally, the extracted information from each constructed dataset is specified, along with the rationale for its selection. Only the selected information was used during the training of the machine learning model.

4.2.1 Criteria

The significant pool of resources regarding APIs provides researchers with a rich choice. Nevertheless, it was simply not feasible to consider a plethora of libraries for a single study. As a result, the first step in the data collection was the composition of criteria. This action aimed to limit the number of APIs to a more feasible sample spanning the duration of the research. Therefore, the criteria (C) we composed and applied are as follows:

C_1 The API must have at least 12,000 usages

C_2 The API must have at most 60 releases

C_3 The API must have been evolving for at least 5 years

C_4 The API must have change log information publicly available

Starting from the first criterion, C_1, we decided that investigating APIs with higher usage would provide more interesting results. This is since these libraries serve a critical function, provided by their widespread use and the dependency of many users on them. Therefore, analysing their evolution could yield more beneficial results. Regarding the second criterion (C_2), we strived to limit the number of releases. Although the analysis was conducted using machine learning, the results still required manual verification, which was time-consuming. The intent was to ensure that our technique worked as expected. C_3 emphasised our focus on libraries that have evolved over a longer period. Five years were selected as a period since it was suspected that such libraries would feature richer patterns of evolution compared to shorter-lived ones. Finally, C_4 required that the change log be publicly available since this has been the main resource for our investigation. Without it, the research would not have been possible. Overall, the criteria options were similarly constructed to those specified by Nikolova [2]. Nevertheless, for C_1 and C_2, higher values were selected to focus on libraries which have a higher degree of evolution.

4.2.2 Java APIs Selection

As addressed in the introduction, the selected libraries were Java-based ones. The main reasons were the abundance of resources on Java APIs, the language's long-standing presence, and its continued widespread usage in commercial projects. Furthermore, as mentioned in chapter 3, Java-based APIs have provoked major research interest. We performed an exploration on the Maven Central Repository (MCR) [12]. MCR was selected because it gathers valuable information, including releases, categories, and usage details. This information could have been valuable for further investigation of each library in more depth. After applying the criteria, we selected a total of six APIs. Four of them, namely **Apache Commons Lang**, **JUnit4**, **Project Lombok**, **Apache Commons IO**, were already featured in the paper by Nikolova [2]. The intention was to use the format, structure and approach as a reference for adding our newly crafted datasets. This would ensure consistency and reduce the time required to compose new datasets. We selected the other two by browsing through the MCR and enforcing the aforementioned criteria. Libraries were chosen based on their popularity and their coverage of domains distinct from those previously mentioned, to yield more diverse and insightful results. This emerged in the selection of **Apache Log4j** and **AppCompat**. All employed libraries, alongside general information for them, are summarised in Table 4.1.

Table 4.1: Selected Java APIs from Maven Central Repository

API	Category	Total Usages	Releases	Changes
Apache Commons Lang	Core Utilities	30,696	19	1,106
Apache Commons IO	I/O Utilities	29,184	22	784
JUnit4	Testing Frameworks & Tools	133,071	19	252
Project Lombok	Code Generators	26,738	57	920
AppCompat	Android Platform & Packages	14,647	47	191
Apache Log4j	Logging Frameworks	19,403	21	588

4.3 Data Extraction

After outlining the approach used to construct the data, this section focuses on the data extraction process. Table 4.2 illustrates the steps and decisions followed for obtaining data. For the **Apache Commons Lang**, **Apache Commons IO**, **JUnit4**, and **Project Lombok** Java APIs, the study by Nikolova [2] already provided datasets which we have utilised for our experiments. For the remaining ones, we aimed at a similar approach, however, omitting certain columns, for the construction of our datasets. Nikolova extracts the information regarding each release log statement in 10-12 columns. Those include information such as *Year*, *Date*, *Aggregated version*, *Version*, *Release*, *Changes*, *By*, *Type*, *General* and *General category*. Although each of these represented useful information, not all were beneficial for a ML model. For example, we reasoned that the person who performed the change does not positively or negatively affect any metric of our model. Therefore, to minimise the manual effort required to create the datasets while still allowing for meaningful data analysis and interpretation of results, we selected six columns. Those are *Date*, *Aggregated version*, *Version*, *Release*, *Changes* and *General category*. For the new software libraries, we utilised the same approach as Nikolova [2]. Thus, extracting changes from release notes or changelogs in spreadsheets. In case of missing data in the MCR, we investigated linked repositories and pull requests for unclear cases.

Table 4.2: Feature selection criteria for the datasets by Nikolova

Column Name	**Purpose**	**Included**	**Justification**
Year	Year of release	✗	The year does not produce useful information for training a ML model.
Date	Concrete date of release	✓	Not applicable for the ML, however useful for analysing changes concerning the period in which they occurred.
Aggregated version	Shows the major version to which the release belongs	✓	Same as the above.
Version	Shows exact version	✓	Same as the above
Release	The type of the release (PATCH, MINOR, MAJOR)	✓	Shows the type of the change and can be related to the impact it has
Changes	Text information regarding the change.	✓	This is the main source for categorising the change for the ML. Thus, it is included.
By	The Responsible person for the change	✗	Does not provide any insights.
Type	Further information regarding the type of the change (not always present)	✗	This has sporadic occurrence, thus it is unreliable for training.
General	Indicates what the change was, e.g. refactoring, handling exceptions, etc.	✗	This information cannot be automatically extracted from a release log text, thus we do not consider using it.
General category	Represents the category of change as listed in Table 4.3	✓	Included as this is a main resource for training the data.

4.4 Data Categorization

After the data was extracted and organised into datasets, we had to decide on one final step in the preparation. Particularly, we were interested in automatically identifying causes of API changes during evolution. Therefore, following the construction of the custom datasets, the next step was to categorise the data. Table 4.3 showcases our categories of changes that were acquired from the paper by Nikolova [2] and further employed for the custom datasets. The labelling from the table was utilised in the datasets to train the machine learning model.

Table 4.3: General categories of changes as analysed in the study by Nikolova [2]

General Category of Changes
Bug fix
Add support for a new feature
Documentation
Maintainability improvement
Update dependency
Performance improvement
Code redesign
Memory improvement
Quality assurance
Deprecate functionality
Support for external development tool
Remove support for external development tool
Security improvement

4.5 Machine Learning Training Process

In the previous section, we explained the construction, extraction and categorisation of our dataset. Once the data had been cleaned, the next step was the training setup for the machine learning model. This was performed in a Jupyter Notebook, where, as in the following subsections, each step of the process is explained in detail. The notebook is accessible through the **GitHub repository**. Figure 4.2 showcases a diagram with the steps performed for the training process and models and tools used. This diagram builds upon the *Training Process* part from Figure 4.1. Figure 4.3 illustrates a flow chart regarding the NLP process involved in our experiment for an example sentence.

4.5.1 Data Exploration

To gain a better understanding of the datasets, we first explored the original data before any preparation or cleaning. The primary goal was to identify potential data inaccuracies and imbalances, as well as to gain an overall understanding of the dataset structures. To achieve this, we examined aspects such as the most frequently occurring words and the length of release messages. The visualisation of these properties was accomplished through a word cloud and bar charts. Additionally, N-grams, collections of n consecutive items that appear in order within a text[1], were examined. The focus was primarily on bi-grams (two-grams), which represent pairs of words that appear adjacent to each other in the text. An example of such from the datasets is a *pull request*.

[1] https://nl.mathworks.com/discovery/ngram.html

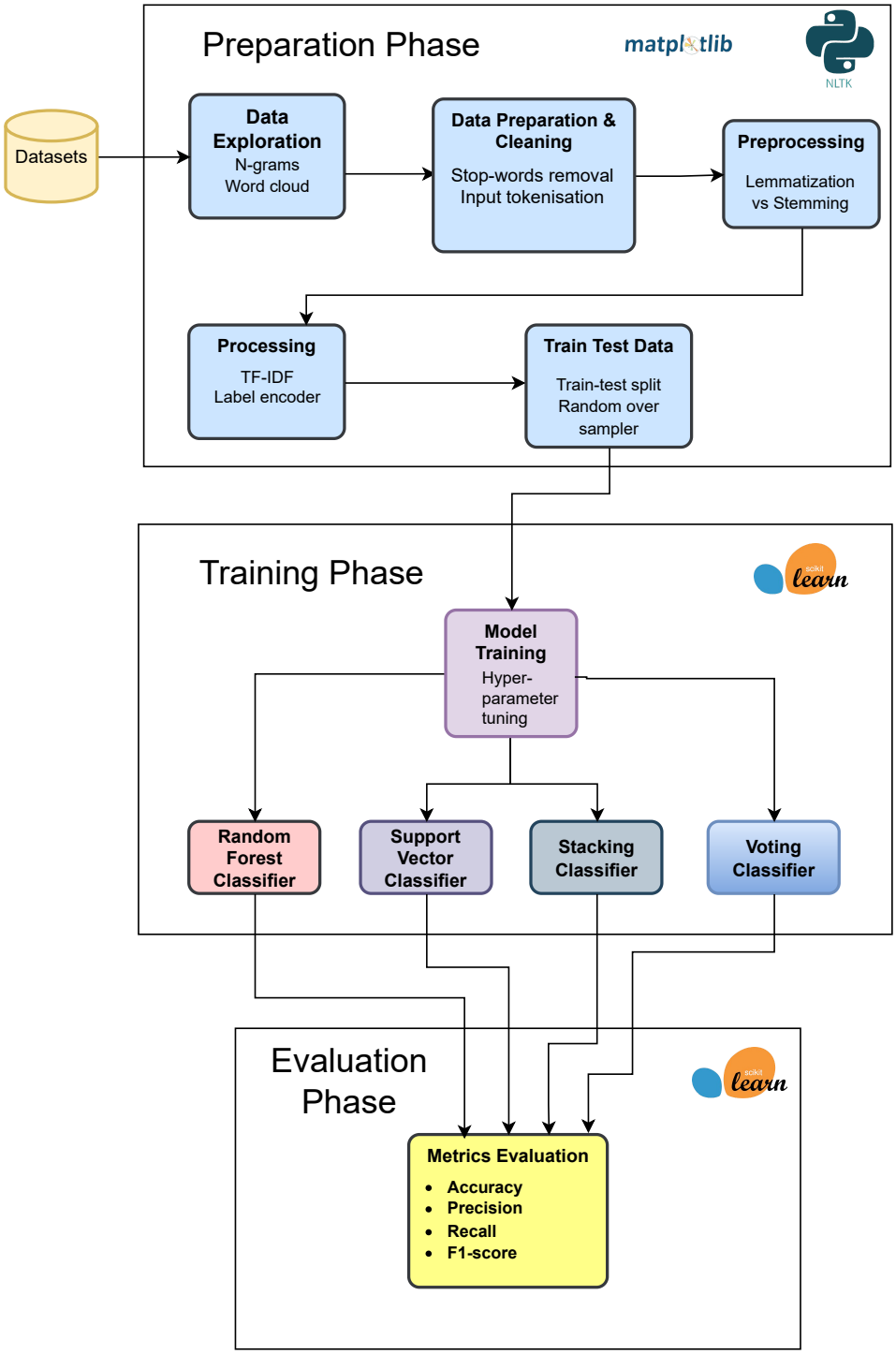


Figure 4.2: Diagram for the complete ML training process

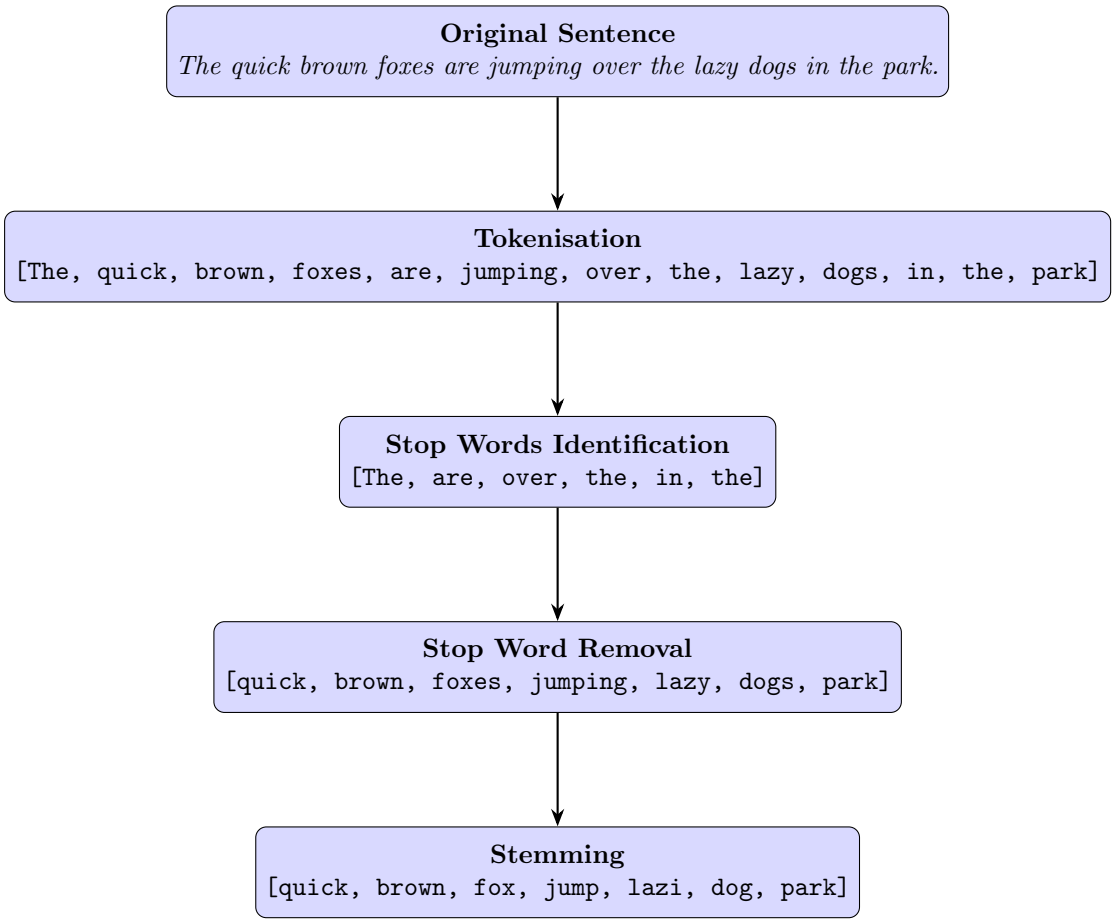


Figure 4.3: Flowchart illustrating the steps in the NLP process for an example sentence

This was a useful technique since some of the N-Grams might directly indicate with high certainty to which category a release log message belongs.

Furthermore, the raw text per release was examined for the word length of a sentence. This information was interesting since it provides insight into the writing style and can showcase whether the authors follow a particular format and pattern of writing. Finally, we examined the target values that were utilised for the categorisation. Based on them, bar and pie charts were plotted to check the distribution of the categories and to check whether the datasets were imbalanced.

4.5.2 Data Preparation & Cleaning

After exploring the data and identifying underlying relationships and patterns, the next step in the training process involved cleaning the dataset. Since the release log messages from the **Changes** column are mainly in the form of sentences, they include common words (stop-words), numbers, and punctuation symbols. Those do not influence the ML model in any way. Therefore, we cleared the data from non-alphabetic characters and stop-words [2] (e.g. *a*, *the*, *is*, *are*). The resulting input per release change was then tokenised. This means that each sentence was organised as a list of the words that construct it. As a result, this prepared the data for the next step, namely, exploring it for possible patterns and relationships.

[2]https://kavita-ganesan.com/what-are-stop-words/

4.5.3 Preprocessing

The input has been cleaned and prepared for the NLP techniques. At this point, all stop-words, punctuation and number symbols have been removed. The next step in the preprocessing requires reducing word variants to one base form so that the inflexions are minimised. This could be achieved by two techniques, lemmatization and stemming. Stemming refers to the action of reducing words to their basic root form [35]. Nevertheless, this does not guarantee that the resulting word would have meaning. For instance, the words *history* and *historical* are both stemmed to *histori*, which is not a word that has any meaning. On the other hand, via the lemmatization a similar action is performed; however, the modified word is an existing normalised form of the word [35]. Therefore, this outputted word should be one that can be found in a dictionary. For instance, *caring* is lemmatized to *care*. To determine the performance of the techniques, we decided to experiment with both stemming and lemmatization. Contrary to our expectations, stemming consistently outperformed lemmatization.

4.5.4 Processing

Once we had explored the preprocessing techniques, the next step was to identify important words. To achieve this, we utilised TF-IDF (Term Frequency-Inverse Document Frequency). The formula for the TF-IDF calculation is listed in Equation 4.3. Essentially, this statistical model evaluates the significance of words in a document pool [36]. Other alternatives, such as Bag of Words, which treats all words equally, also exist. However, given that for certain releases, some words (e.g. *Bug*) are more important than others, we identified TF-IDF as a more suitable option. Analysing words in ML requires converting them to a numerical format in the form of a matrix or vector. We combined the tokens from per release log and performed an operation called *fit_transform*, which converts the combined tokens into a matrix of TF-IDF features. These processed values could then be utilised as input for the ML models during both the training and testing phases. The final step required the labelling or codification of the target variables. Those were the categories of changes. For that, we utilised an *LabelEncoder* which consistently provided a one-to-one mapping between a category and a number.

$$\mathrm{TF}(t,d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d} \tag{4.1}$$

$$\mathrm{IDF}(t,D) = \log\left(\frac{|D|}{|\{d \in D : t \in d\}|}\right) \tag{4.2}$$

$$\text{TF-IDF}(t,d,D) = \mathrm{TF}(t,d) \times \mathrm{IDF}(t,D) \tag{4.3}$$

4.5.5 Machine Learning Setup

To this end, we have performed all the pre- and post-processing techniques concerning the text data. The next part required setup for testing and measures to handle the class imbalance that was identified. Regarding the train-test split, we performed one using the train_test_split() method from scikit-learn (sklearn). The proportion was **80/20** in favour of training to ensure a representative sample for the testing data. Additionally, a $random\_state = 42$ was selected as a parameter. The *random_state* was configured to ensure that the data split remains consistent and reproducible across runs on the same dataset. After splitting the data, we confirmed the class imbalance between certain categories. For instance, in the **JUnit** dataset, some categories contained only one instance, such as *Security improvement*, while others (e.g. *Code redesign*) had 36. To mitigate this, a resampling was needed. Since the overall datasets were small (the largest one had 1,106 and

the smallest 191), we decided to run the **RandomOverSampler** [37]. It works by randomly selecting and duplicating existing samples from the minority class in the training dataset to balance the class distribution.

After completion of training and resampling setup, the next step was to prepare for the model training. Since the space for model parameters is large and infeasible to explore manually, it was decided to perform hyper-parameter tuning utilising **GridSearchCV** from scikit-learn (sklearn) [38]. For that, we passed different parameters to the GridSearchCV, depending on the examined ML model. An alternative was **RandomizedSearchCV** that has a drastically slower run time. However, *GridSearchCV* performs an exhaustive search over the specified parameter space, which helps to optimise the performance of the ML model. This comprehensive approach is the primary reason for its selection. Finally, *StratifiedKFold* was specified as the cross-validation strategy for *GridSearchCV* to evaluate the performance of different parameter configurations. This approach ensures that during the hyper-parameter tuning, each fold maintains the same class distribution as the overall training dataset, leading to more reliable and balanced validation results.

4.5.6 Machine Learning Techniques

Performing the machine learning experiment required the utilisation of several ML and NLP techniques. In the previous subsections, we outlined the general process for the experiment. Particularly, the NLP steps were listed, alongside their purpose. Regarding the ML, the test-training split, sampling and hyperparameter tuning were explained. With respect to the ML techniques, supervised learning algorithms were selected. Additionally, each change had to be classified by one of the categories mentioned in Table 4.3. This required research on supervised classification models. In developing the classification model, several experiments were conducted using different algorithms that were promising. This was done with consideration of the dataset's characteristics, including the presence of 11 classes, a relatively small number of text entries, and class imbalance.

Overall, for each dataset, at least 25 ML models were explored with a combination of different parameters. Only a handful of those produced results with higher accuracy. The main reason was the nature of the data itself. As mentioned before, it consisted of imbalanced and small data. Based on the results, classifiers chosen for this study include **Random Forest Classifier (RFC)**, **Support Vector Classifier (SVC)**, **Voting Classifier (VC)**, and **Stacking Classifier (SC)**, all of which are available on the sklearn library. In the following subsection, we will provide a detailed explanation of how each classifier operates. The rationale behind their selection for the experimental setup, their respective advantages and disadvantages, also the parameters passed to each model. All the information is acquired from the official documentation available on sklearn [39].

Random Forest Classifier

The RFC is an ML model which trains multiple decision tree classifiers on different sub-samples of the training dataset. The idea is to then average the predictions, which would enhance the accuracy and reduce overfitting [39]. The primary reason to select the Random Forest Classifier (RFC) was its ability to provide a robust and versatile model that reduces the risk of overfitting and performs effectively on imbalanced datasets. With regards to the shortcomings, it is computationally expensive, which is not problematic given the size of our datasets. Nevertheless, for larger ones, it can pose challenges. Moving to the parameters, we have experimented with different values in the Jupyter notebook. In the next six lines, we list the explanation for each parameter, obtained from the sklearn documentation [39]:

- *criterion* - function to measure the quality of a split
- *n_estimators* - the number of trees in the forest
- *max_depth* - maximum depth of a tree

- *min_samples_leaf* - minimum number of samples needed at a leaf node
- *class_weight* - weights per class/category in the dataset
- *random_state* - utilised to manipulate the randomness of the model

Support Vector Classifier

The second model that we considered was SVC. It is a variant of Support Vector Machine (SVM) that, similarly to the latter, identifies the hyperplane which separates the data points of different classes with the highest accuracy [40]. As a result, the SVC is especially useful in high-dimensional datasets and a robust choice for multi-class classification tasks. On the other hand, it is sensitive to noise and not scalable to larger datasets. Since both aspects are managed in the assessment, they do not pose a threat to our experiment. The parameters that were utilised are as follows:

- *C* - controls the training loss for misclassified data
- *gamma* - coefficient for the kernel
- *kernel* - select kernel type for the model
- *probability* - controls enablement of probability estimates
- *max_iter* - maximum limit for iterations
- *class_weight* - weights per class/category in the dataset
- *random_state* - utilised to manipulate the randomness of the model

Voting Classifier

A different classifier from the aforementioned two is the Voting classifier. It combines the predictions of multiple individual models that enable it to output a final prediction [39]. This aids in deciding which algorithm is the best performing, according to the provided dataset. The models are trained independently on the training dataset, and afterwards, while performing a prediction, a majority rule is utilised to decide the final prediction. For our research, VC with SVC, RFC and Logistic Regression were selected. It is important to note that the Logistic Regression model is not limited to regression tasks; it also supports classification tasks.

Stacking Classifier

Finally, a model similar to the Voting classifier is the Stacking. It utilises multiple classifiers. However, combining the predictions of these individual estimators reduces their individual biases [39]. Specifically, the predictions established by each estimator are ***stacked*** and fed as an input for a final estimator, which computes the overall prediction. This model is particularly effective in handling complex relationships by providing a flexible configuration.

4.5.7 Metrics

Assessing the performance of the aforementioned machine learning models required the selection of appropriate evaluation metrics. Therefore, a well-established set of them was selected to assess the effectiveness of the ML models. In chapter 2, we outlined the foundational evaluation metrics for ML models, alongside the notion and purpose of confusion matrices. We then explained the meaning and implications of the accuracy, precision, recall, and F1-score metrics in the context of evaluating ML models.

By incorporating all relevant measurements in the data analysis, the trade-offs between specific metric values and the characteristics of the dataset were effectively considered. Consequently, this aided in providing insights for interpreting the results. Given the multi-class nature of our dataset

and the ML techniques used, we need an effective way to compute the average results for each class based on the metrics obtained. To address this, we focus on two types of averaging: the *macro* average and the *weighted* average, shown in Equation 4.4 and Equation 4.5. These approaches allow for a more comprehensive model performance evaluation across all classes. The weighted average for the $F1$-score computes the score for each class, and then the average is acquired by assigning weights proportional to the representation of the class [41]. Therefore, the $F1$-score of a category with 150 samples has a higher impact on the average score, compared to the one with 20 samples. On the other hand, the macro average calculates the unweighted mean of all classes in the dataset [41]. As a result, it ignores the importance of popular classes, since it treats all of them equally. In Equation 4.4 and Equation 4.5, the formulas for both F1-score variants are demonstrated, where N is the number of classes and w_n indicates the proportion of classes belonging to the n class. We incorporated both approaches in the experiment to ensure a more comprehensive analysis.

$$\mathrm{F1}_{\mathrm{macro}} = \frac{1}{N}\sum_{n=1}^{N}\mathrm{F1}_n \tag{4.4}$$

$$\mathrm{F1}_{\mathrm{weighted}} = \sum_{n=1}^{N} w_n \cdot \mathrm{F1}_n \tag{4.5}$$

4.5.8 Ground Truth

Evaluating the metrics required identifying the ground truth for each release log statement. To achieve this, we adopted the predefined categories used for classifying release log messages, originally introduced in the work by Nikolova [2]. To maintain consistency, the same categories were applied when labelling our manually created datasets (Log4j and AppCompat). All categories are listed in Table 4.3. To further ensure reliability, both Nikolova's datasets and our custom-labelled ones were reviewed multiple times for consistency in labelling.

4.6 Architecture Analysis

To answer RQ_2, we aimed to investigate the architecture and functionalities of three of the libraries, namely JUnit, Apache Commons IO and Project Lombok. We selected them since they exhibit interesting behaviour and are feasible for research for the provided period. On the other hand, Log4j, for instance, has a codebase where some releases contain over 5,000 methods. This was posing a significant challenge for empirical analysis and demanding substantial computational resources to generate visualisations like dependency graphs. It is crucial to specify that not all codebases from the release notes versions were available. While for Apache Commons IO and Project Lombok, we found all versions covered in the analysis by Nikolova [2], for JUnit, the codebases from 4.0 to 4.5 were available in the official repository. One reason is that those versions cover changes from 2006-2008 and were probably deprecated when the organisation decided to move to GitHub, where the newer versions are stored.

Considering the experimental work we performed, it can be divided into three stages. First, we investigated the overall codebase and releases of the selected libraries. As explained in the background chapter, we explored the most common categories, application of design patterns and metrics of the codebase version to obtain an insight into what to expect. For the metrics, we utilised the plugin MetricsReloaded [42] from IntelliJ, specifically calculating *source code lines*, *number of packages*, *number of classes* and *number of methods*. Second, based on the information from the paper by Nikolova and the codebase for the corresponding releases, we attempted to check empirically what has been modified. Then, using the techniques and findings from RQ_1, we analysed the release log changes. The focus was on those, impacted the architecture and functionalities with the aim of validating whether such changes are presented in the release log messages. The third fold of the RQ_2 experiment involved evaluating the effects of architectural changes during the evolution phase. To acquire this information, we employed tools designed for visualising class-level architecture. Consequently, tools such as *javapers* [1], arcana [43], *bubbleTea 2.0* [44], and class diagram reconstruction were selected to support the process. The subsequent

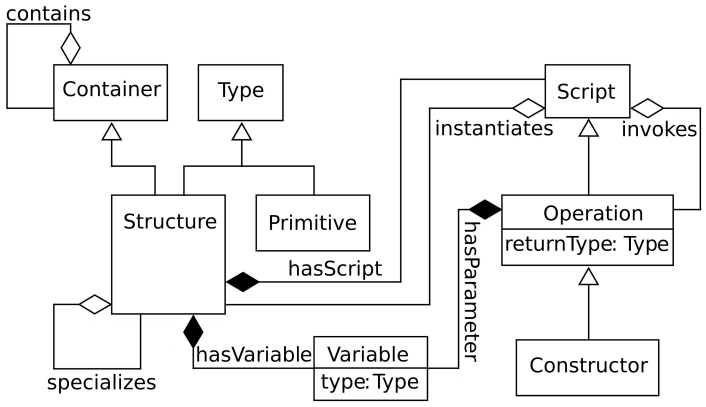


Figure 4.4: Example knowledge graph obtained from source code [1]

paragraph provides an overview of these tools, along with justification for how each contributes to achieving the research objectives.

Starting from the metrics tool, it enables the measurement of various software metrics, including the number of classes, test coverage, cyclomatic complexity and more. To elaborate, it is a useful and valuable tool for investigating quantifiable software characteristics to form an impression regarding the codebase. Moving to javapers and arcana, both are fundamental instruments for preparing the input to bubbleTea. To be more precise, javapers extracts the knowledge graph from a provided source code. An example of that is illustrated in Figure 4.4. After obtaining the corresponding graph in a JSON format, the next step was to apply the arcana tool to the knowledge graph. As a result, arcana would perform analysis such as dependency profile, including a large language model API which would examine and enrich graph nodes by adding properties such as description, role stereotype and layer [43]. Based on the prework performed, the output from arcana is then uploaded to bubbleTea, which visualises the information in a user-friendly way, illustrating the package and class-level meta-information details.

The class and package diagram reconstruction offered by the JetBrains diagramming tool [45] featured an additional visualisation technique that significantly enhanced the analysis process. Particularly, it aided in comparing the class-level structure changes between two or more release versions. This enabled an empirical investigation of more specific modifications introduced between the two versions under examination. This approach allowed us to track alterations in the codebase to assess their impact closely. Ultimately, this enhanced and facilitated one of the primary goals of the research. Those were to determine whether the key changes, both at the architectural and functional levels, were adequately represented in the release log messages, or whether they were neglected or omitted entirely.

4.7 Tools

Due to the data science/machine learning nature of this research, our preferred language was Python. It has been consistently ranked high as one of the most popular languages in the last eight years [7]. Partially the reason is its versatility, simplicity and readability. In data science, libraries such as Numpy, Matplotlib, Scikit-Learn, Pandas and NLTK [46] provide useful functionalities that facilitate the development process. We utilise most of these libraries during the implementation of our techniques. The tools for RQ_2 were listed and explained in the previous section. Therefore, we would not provide redundant information here.

In the whole chapter 4, we outlined the key methodology-related decisions for our experiments, along with their respective advantages and disadvantages. To check the practical matters concerning the experiments on RQ_1, visit this GitHub repository. There is the whole code utilised for the experiments.

Chapter 5

Results & Discussion

In this chapter, the results and discussion regarding the research questions are presented. We systematically analyse the datasets, highlighting any imbalances and potential biases. Next, we showcase the outcomes from the machine learning models, providing a comprehensive overview of their performance. Consequently, we present the findings, examining the advantages and broader implications related to the research questions. The final part answers all the research questions based on the obtained results and insights from the discussion. Since including all diagrams in this chapter was not feasible, those not presented here can be found in the Appendix or the GitHub repository.

5.1 Data Distribution & Analysis

Before delving into the results obtained by the ML models, we explore the datasets. To display them, bar plots and word clouds are presented. These visuals were selected since they clearly illustrate the distribution of the datasets' categories and the most common words in the **Changes** column (which contains the release log messages). Figure 5.1 to Figure 5.4 display the bar plots for the selected datasets. Overall, as evident from the visuals, some categories are dominating, such as *Code redesign*, *Bug fix* and *Adding support for a new feature*. This immediately poses a challenge to the ML models. They have difficulty classifying release log messages from categories with a small sample size, e.g., fewer than ten. The bar plots illustrate the distribution of classes, and from them, it can be concluded that all datasets are highly imbalanced. Moving to the word cloud Figure 5.5, it signifies the most common words occurring in the **Changes** column. Among them is *fix*, which our empirical analysis shows mostly refers to a *Bug fix*. Therefore, tuples such as this could contain highly biased information regarding the possible category of change. For the JUnit dataset, *Pull* is the most common word, which is part of almost every release message. It is connected with the expression *Pull request*, specifying the Jira issue that has been resolved. Under these circumstances, it holds little significance for an ML model. Similarly to *Gary Gregory* in Figure 5.6, the name appears in almost every release log message, since the person is a prominent maintainer of the Apache Commons IO API. Figure 5.7 and Figure 5.8 illustrate the word distributions within the *Changes* column for the JUnit and Project Lombok datasets, respectively. From the graphs, it is evident that Project Lombok utilises, on average, more words to explain the performed change compared to JUnit. Overall, based on data exploration, it is clear that we are working with highly imbalanced multi-class data.

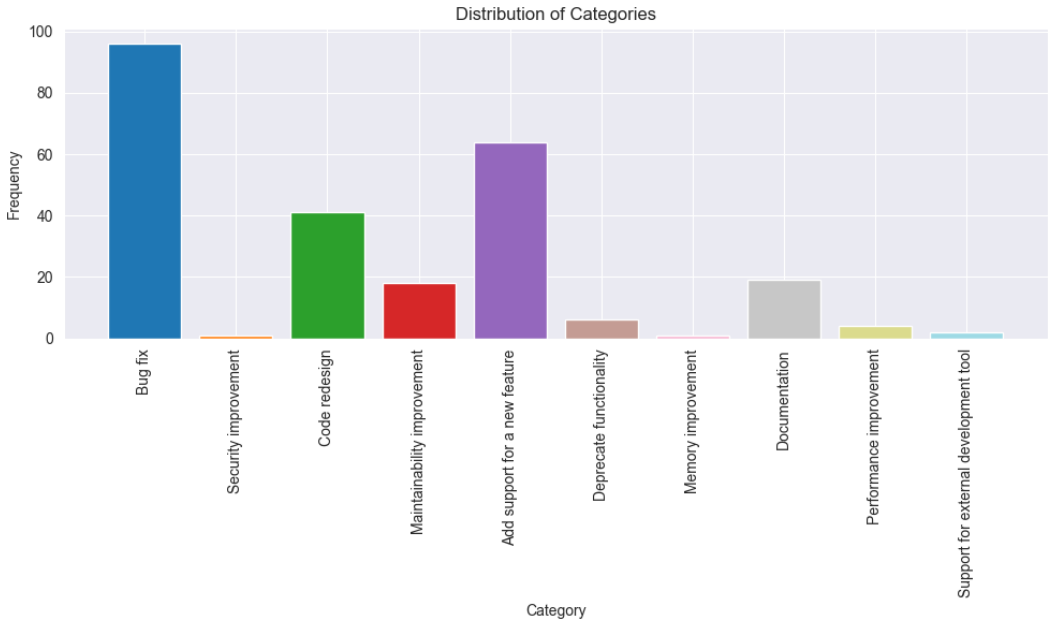


Figure 5.1: Bar chart distribution for categories of JUnit dataset.

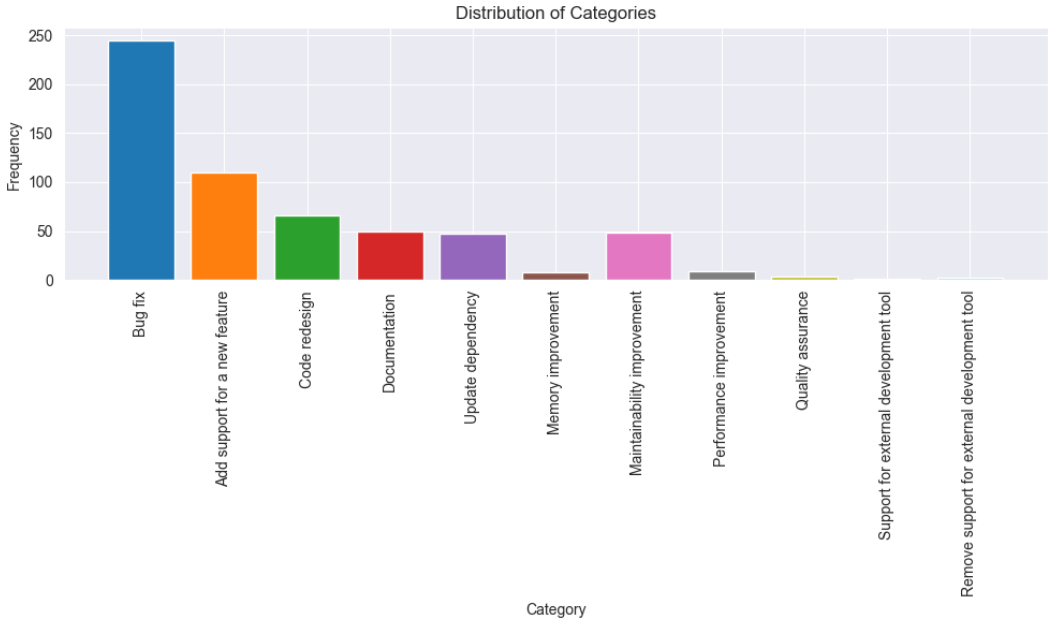


Figure 5.2: Bar chart distribution for categories of Log4j dataset.

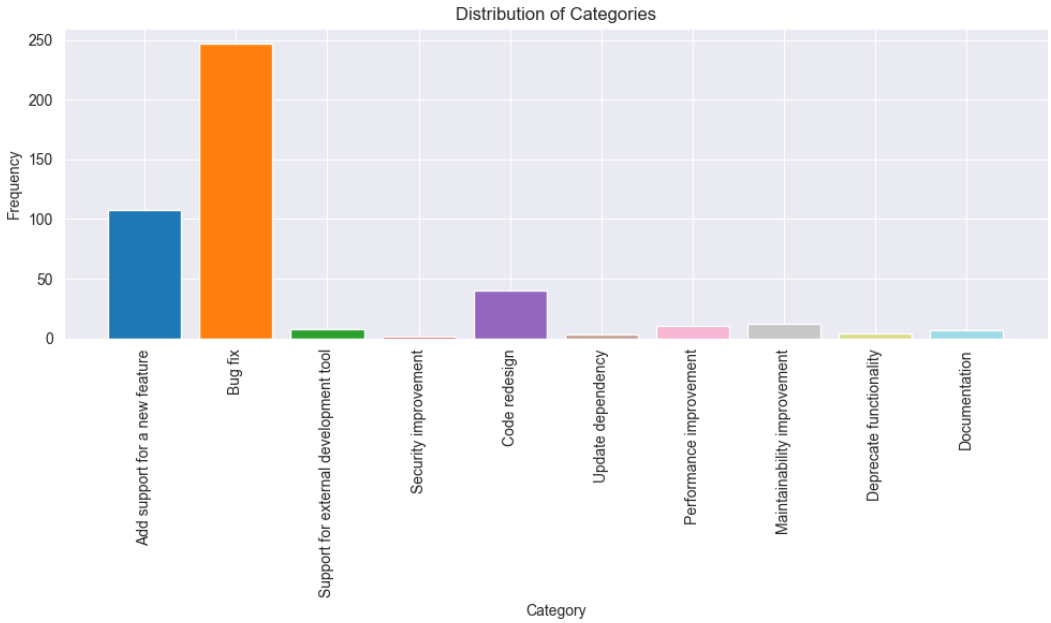


Figure 5.3: Bar chart distribution for categories of Project Lombok dataset.

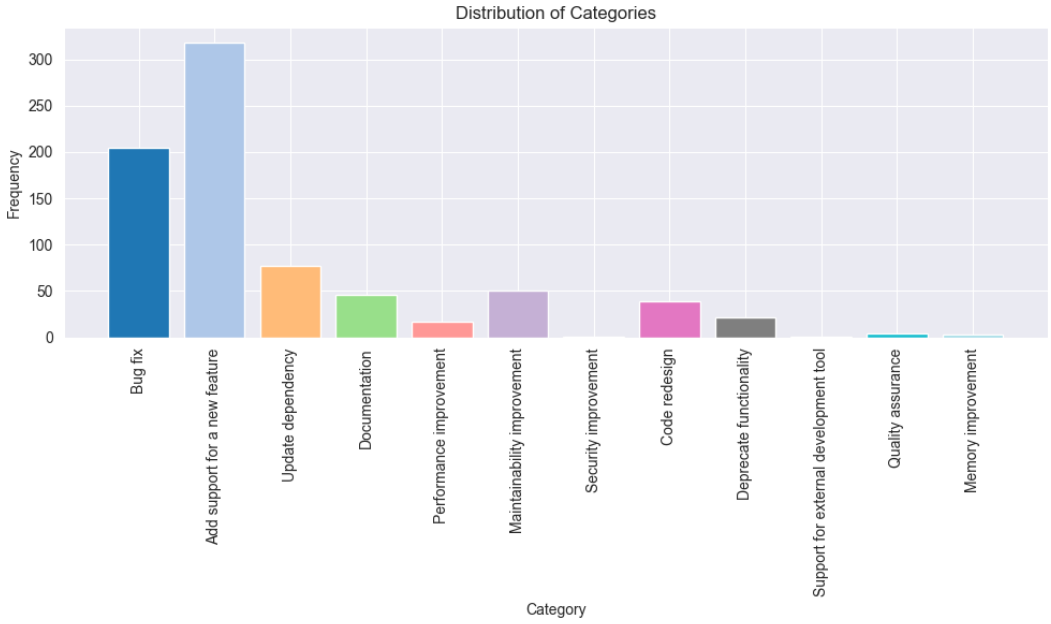


Figure 5.4: Bar chart distribution for categories of Apache Commons IO dataset.

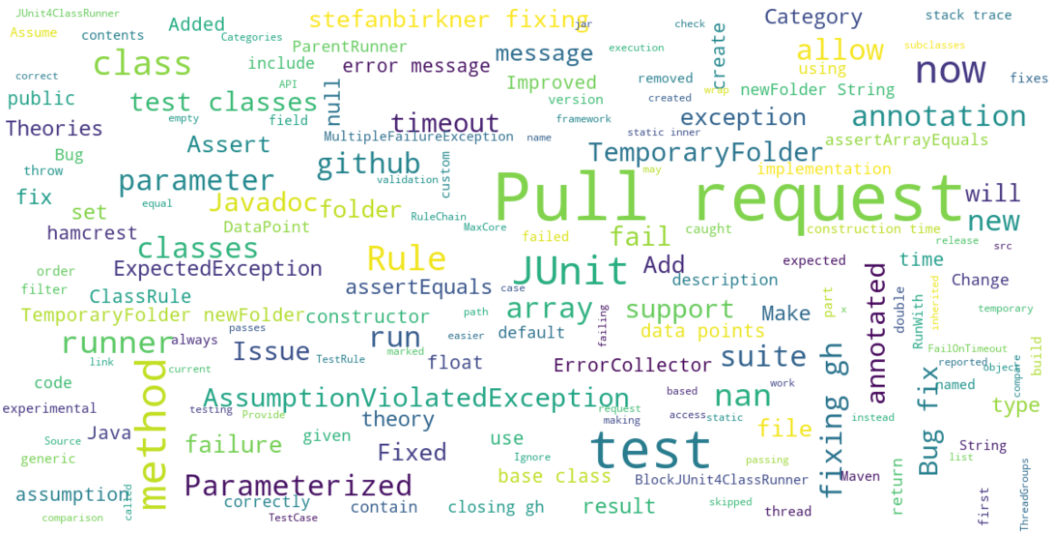


Figure 5.5: Word cloud for JUnit dataset

Figure 5.6: Word cloud for Apache Commons IO dataset

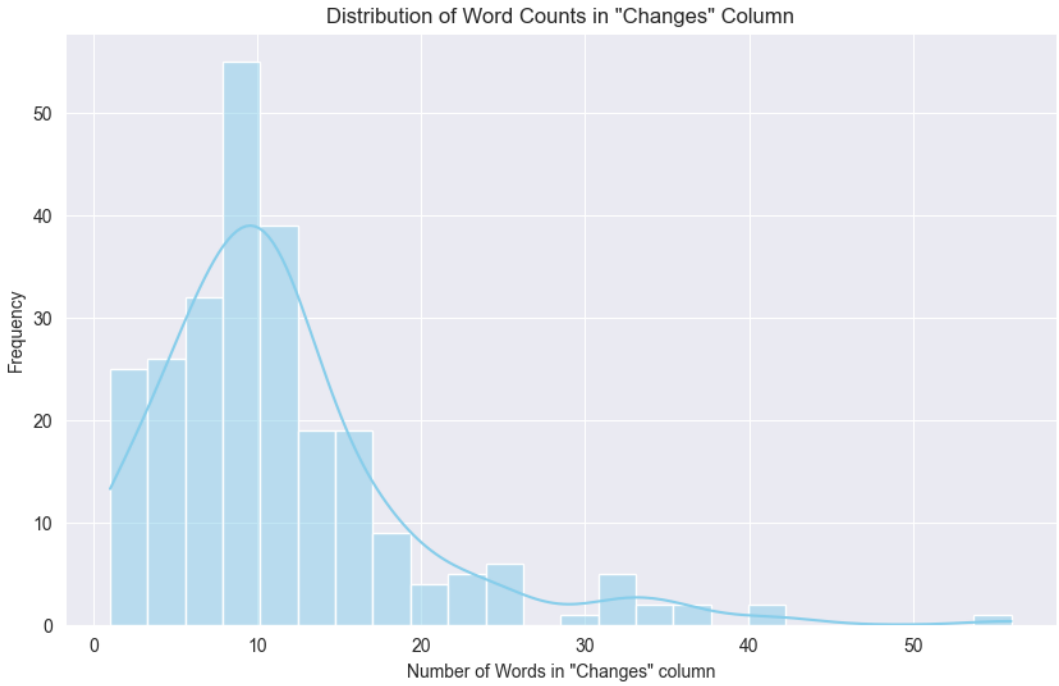


Figure 5.7: Text count distribution in Changes column for JUnit dataset

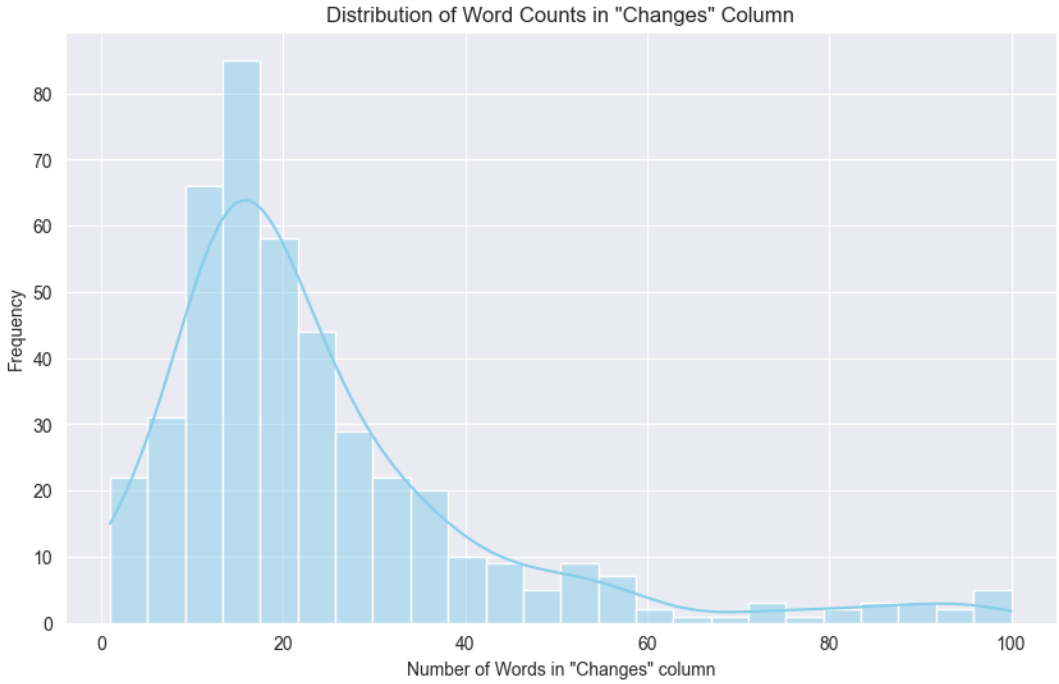


Figure 5.8: Text count distribution in Changes column for Project Lombok dataset

5.2 Automatic Categorization

5.2.1 Results

In this subsection, we share the results related to RQ_1, using the approach identified in chapter 4. We illustrate the results via a combination of confusion matrices and classification report elements. This information aids in visualising the performance for a particular dataset based on category level and also as a whole. To keep the document clean, we include some of the figures and tables in this subsection. Results selection is based on the best-performing machine learning models across the selected datasets. Furthermore, we highlighted instances demonstrating behaviour particularly relevant to the research objectives. The remaining visualisations are provided in the Appendix or the GitHub repository. For the current section, the performance of ML models on **JUnit**, **Log4j**, **Apache Commons IO**, and **Project Lombok** datasets is shared.

Confusion Matrices

Figure 5.9 to Figure 5.12 display four confusion matrices from the aforementioned datasets. Overall, we have concentrated on the models deemed as most intriguing and best-performing on the given dataset. Figure 5.9 displays the performance for each of the categories on the Apache Commons IO dataset. Here we can witness that the ML model accuracy for *Add support for a new feature*, *Documentation*, and *Update dependency* is high. For the remaining, the performance is fluctuating. Figure 5.10 exhibits similar properties, with the main difference being in the *Bug fix* category. It has a larger sample size compared to Apache Commons IO and higher accuracy. Moving to Project Lombok and JUnit, represented in Figure 5.11 and Figure 5.12, the trends are similar. ML categorisation on *Add support for a new feature* and *Bug fix* is stable and high. However, for the JUnit dataset, the accuracy on the *Documentation* is low, compared to Log4j and Apache Commons IO.

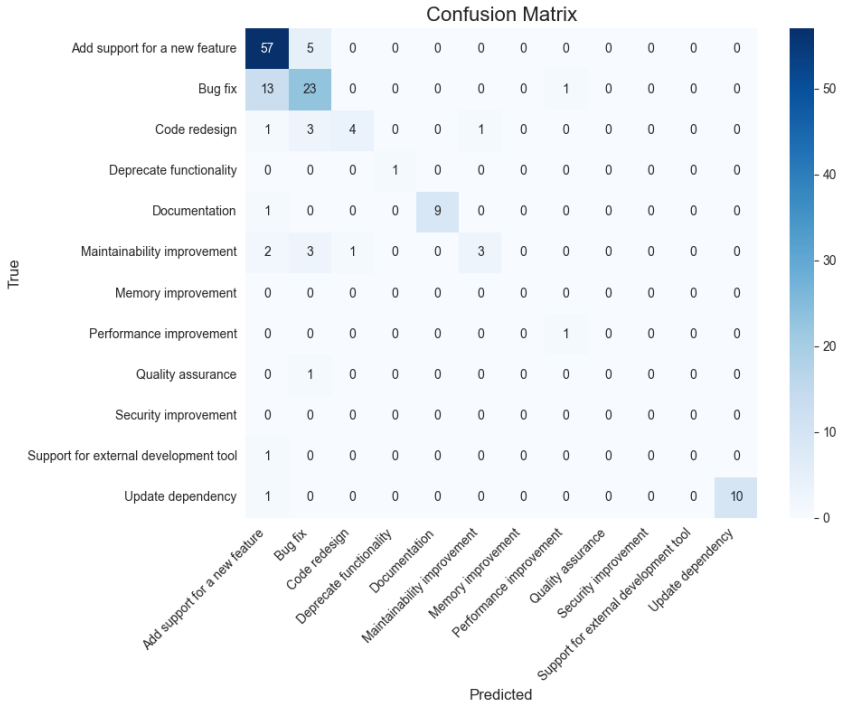


Figure 5.9: Apache Commons IO - confusion matrix for Random Forest Classifier

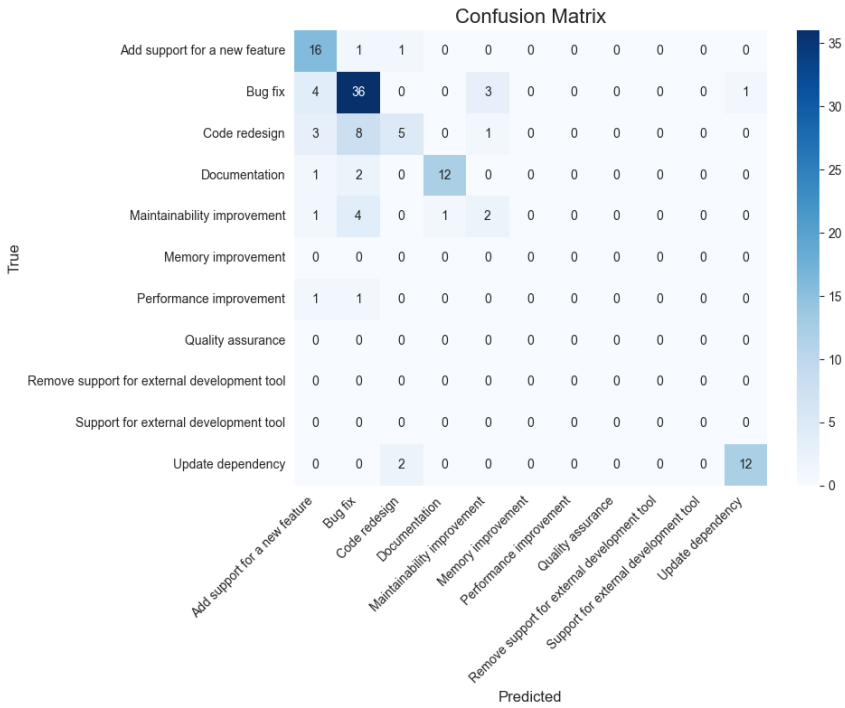


Figure 5.10: Log4j - confusion matrix for Voting Classifier

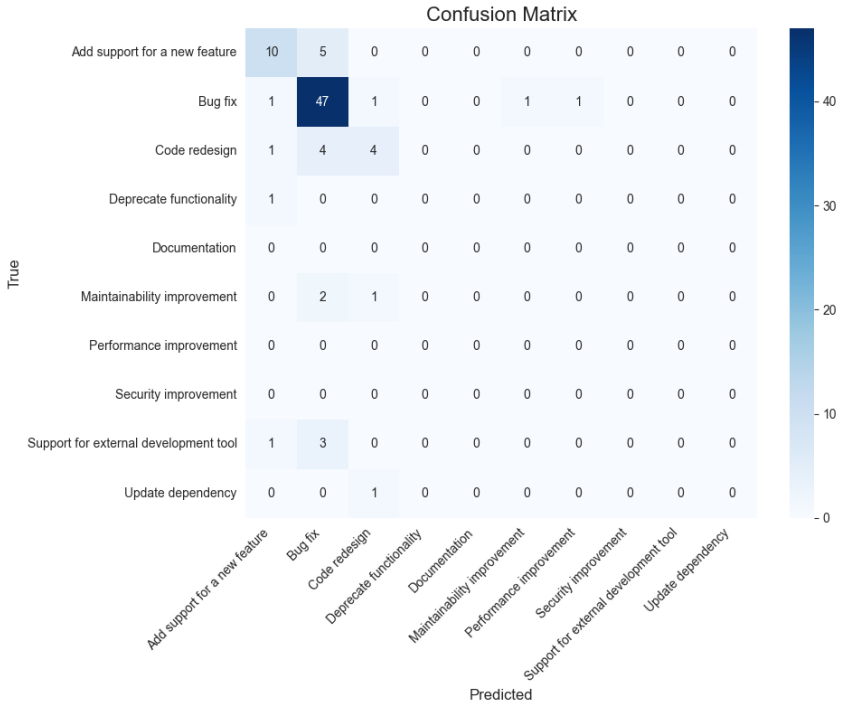


Figure 5.11: Project Lombok - confusion matrix for Random Forest Classifier

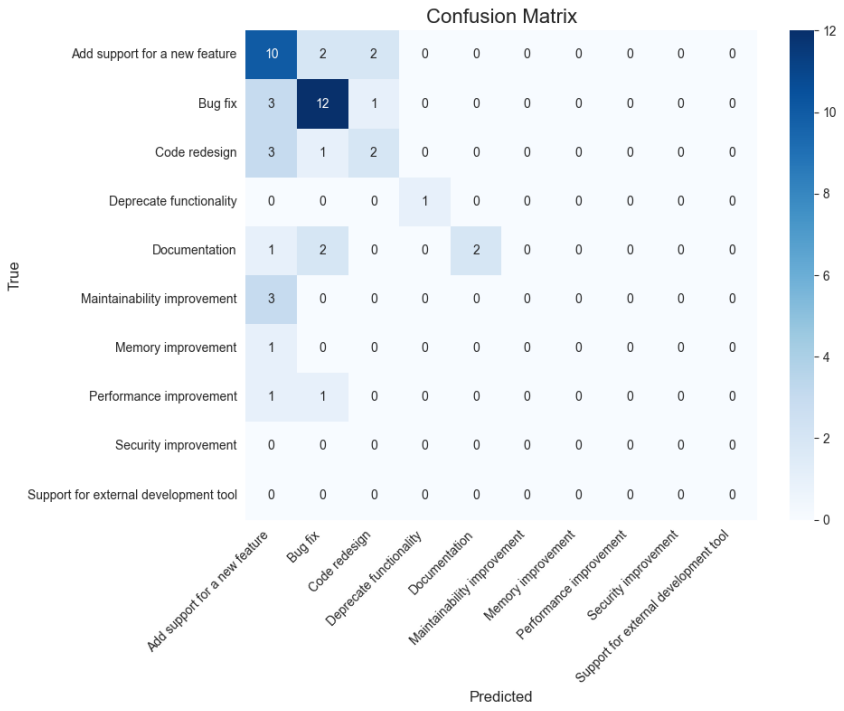


Figure 5.12: JUnit - confusion matrix for Random Forest Classifier

Classification Reports

We provide the classification reports for the aforementioned models. These reports would aid in understanding the performance in-depth. They illustrate the evaluation metrics of the ML model per category, including accuracy, recall, precision, and F1-score. The best-performing classifiers per dataset are depicted in the tables, selected based on accuracy and F1-score. In the tables below, the *Support* column indicates the number of test instances utilised to evaluate the performance of the ML classification models. Moreover, we designate two colours for highlighting interesting categories. Categories with underwhelming performance are highlighted in darker grey. The brighter grey is for satisfactory-performing rows.

Table 5.1: Classification report for Apache Commons IO employing the Random Forest Classifier

Category	**Precision**	**Recall**	**F1-score**	**Accuracy**	**Support**
Add support for a new feature	0.750	0.919	0.826	-	62
Bug fix	0.657	0.622	0.639	-	37
Code redesign	0.800	0.444	0.571	-	9
Deprecate functionality	1.000	1.000	1.000	-	1
Documentation	1.000	0.900	0.947	-	10
Maintainability improvement	0.750	0.333	0.462	-	9
Performance improvement	0.500	1.000	0.667	-	1
Quality assurance	0.000	0.000	0.000	-	1
Support for external development tool	0.000	0.000	0.000	-	1
Update dependency	1.000	0.909	0.952	-	11
Global Metrics					
Overall Accuracy	-	-	-	0.761	142
Macro Average	0.538	0.511	0.505	-	142
Weighted Average	0.755	0.761	0.745	-	142

Considering Table 5.1 for the Apache Commons IO dataset, the accuracy is above the 70% threshold. This demonstrates that, in general, the ML model has a high degree of correct categorisation, particularly for categories with larger sample sizes. The *Code redesign* and *Maintainability improvement* categories show lower recall and F1-score results. However, due to their small sample sizes, their impact on performance metrics is limited.

Table 5.2: Classification report for JUnit employing the Random Forest Classifier

Category	**Precision**	**Recall**	**F1-score**	**Accuracy**	**Support**
Add support for a new feature	0.455	0.714	0.556	-	14
Bug fix	0.667	0.750	0.706	-	16
Code redesign	0.400	0.333	0.364	-	6
Deprecate functionality	1.000	1.000	1.000	-	1
Documentation	1.000	0.400	0.571	-	5
Maintainability improvement	0.000	0.000	0.000	-	3
Memory improvement	0.000	0.000	0.000	-	1
Performance improvement	0.000	0.000	0.000	-	2
Global Metrics					
Overall Accuracy	-	-	-	0.562	48
Macro Average	0.352	0.320	0.320	-	48
Weighted Average	0.530	0.562	0.523	-	48

Based on the result from Table 5.2, a notable difference between the JUnit and Apache Commons IO datasets is the number of elements each contains. On the one hand, the former consists of 48 test samples, whereas the latter consists of 142. The ML model exhibits poor performance on the JUnit dataset as a whole, with lower macro and weighted average metrics, as well as reduced scores across individual categories, compared to Apache Commons IO. As a result, the 70% accuracy threshold is not satisfied, with the RFC model achieving a low accuracy of 56%.

Table 5.3: Classification report for Log4j employing the Voting Classifier

Category	**Precision**	**Recall**	**F1-score**	**Accuracy**	**Support**
Add support for a new feature	0.615	0.889	0.727	-	18
Bug fix	0.692	0.818	0.750	-	44
Code redesign	0.625	0.294	0.400	-	17
Documentation	0.923	0.800	0.857	-	15
Maintainability improvement	0.333	0.250	0.286	-	8
Performance improvement	0.000	0.000	0.000	-	2
Update dependency	0.923	0.857	0.889	-	14
Global Metrics					
Overall Accuracy	-	-	-	0.703	118
Macro Average	0.374	0.355	0.355	-	118
Weighted Average	0.692	0.703	0.682	-	118

As shown in Table 5.3, the Voting Classifier has stable performance on almost all categories in the Log4j dataset except for *Maintainability improvement* and *Performance improvement*. The model again surpasses the 70% accuracy threshold. However, compared to the previous tables, the macro average scores are approximately half the weighted average scores.

Table 5.4: Classification report for Project Lombok employing the Random Forest Classifier

Category	**Precision**	**Recall**	**F1-score**	**Accuracy**	**Support**
Add support for a new feature	0.714	0.667	0.690	-	15
Bug fix	0.770	0.922	0.839	-	51
Code redesign	0.571	0.444	0.500	-	9
Deprecate functionality	0.000	0.000	0.000	-	1
Maintainability improvement	0.000	0.000	0.000	-	3
Support for external development tool	0.000	0.000	0.000	-	4
Update dependency	0.000	0.000	0.000	-	1
Global Metrics					
Overall Accuracy	-	-	-	0.726	84
Macro Average	0.206	0.203	0.203	-	84
Weighted Average	0.657	0.726	0.686	-	84

From Table 5.4, it can be observed that the RFC model achieves an F1-score above 65% and precision above 70% for the *Add support for a new feature* and *Bug fix* categories in the Project Lombok dataset. Contrarily, the remaining categories show poor classification performance, with low recall and precision for *Code redesign*, and no correct classifications for the other four categories. Due to the absence of correct classifications in four categories, the macro average scores are roughly one-third of the weighted ones. Nevertheless, the threshold of 70% is achieved.

Table 5.5: Classification report for the combined dataset featuring entries from Project Lombok, Apache Commons IO, JUnit and Log4j employing Random Forest Classifier

Category	Precision	Recall	F1-score	Accuracy	Support
Add support for a new feature	0.775	0.892	0.829	-	120
Bug fix	0.761	0.781	0.771	-	155
Code redesign	0.481	0.310	0.377	-	42
Deprecate functionality	0.500	1.000	0.667	-	3
Documentation	0.792	0.864	0.826	-	22
Maintainability improvement	0.538	0.333	0.412	-	21
Memory improvement	1.000	1.000	1.000	-	1
Performance improvement	0.500	0.429	0.462	-	7
Quality assurance	0.000	0.000	0.000	-	1
Support for external development tool	0.000	0.000	0.000	-	4
Update dependency	0.773	0.850	0.810	-	20
Global Metrics					
Overall accuracy	-	-	-	0.735	396
Macro average	0.471	0.497	0.473	-	396
Weighted average	0.711	0.735	0.717	-	396

The performance of the RFC model on the combined dataset is satisfactory, particularly for categories with larger sample sizes, as demonstrated in Table 5.5. The model achieves high results on *Add support for a new feature* and *Bug fix* with above 75% results on all evaluation metrics. However, the results on *Maintainability improvement* and *Performance improvement* remain comparatively low. Interestingly, despite the higher support count, *Code redesign* still yields lower metric values. Such patterns were previously noted within the individual datasets.

Table 5.6: Classification report for combined dataset featuring the entries from Project Lombok, Apache Commons IO, JUnit and Log4j employing Voting Classifier

Category	Precision	Recall	F1-score	Accuracy	Support
Add support for a new feature	0.773	0.850	0.810	-	120
Bug fix	0.709	0.865	0.779	-	155
Code redesign	0.588	0.238	0.339	-	42
Deprecate functionality	0.500	0.667	0.571	-	3
Documentation	0.889	0.727	0.800	-	22
Maintainability improvement	0.667	0.381	0.485	-	21
Memory improvement	1.000	1.000	1.000	-	1
Performance improvement	0.667	0.286	0.400	-	7
Quality assurance	0.000	0.000	0.000	-	1
Support for external development tool	0.000	0.000	0.000	-	4
Update dependency	0.842	0.800	0.821	-	20
Global Metrics					
Overall accuracy	-	-	-	0.735	396
Macro average	0.510	0.447	0.462	-	396
Weighted average	0.719	0.735	0.712	-	396

The VC has almost identical scoring for the combined dataset as the RFC one, as evident from

Table 5.6. A notable difference is related to the *Code redesign*, *Maintainability improvement* and *Performance improvement* categories, where the VC achieves higher evaluation scores compared to the RFC model. These improvements are predominantly reflected in the macro average, where VC outperforms the RFC model. Nevertheless, the overall accuracy for both RFC and VC is 73.5%, exceeding the 70% threshold

5.2.2 Discussion

In this subsection, we analyse and interpret the results obtained from the experiments. The primary focus is concentrated on identifying significant relationships and explaining how the characteristics of a dataset affect the performance of ML models.

Overall Performance

Starting with performance, in Table 5.1 through Table 5.6, shows that the overall accuracy of five out of six models remains stable. It is concentrated in the range of 0.703 - 0.761, indicating a margin of around 4.5%, depending on the model. From this, we can claim that the general performance of the ML models is satisfactory and can be trusted for categorising changes. Nevertheless, we can infer that one dataset performs unsatisfactorily on all metrics, namely JUnit. This implies that these problematic properties are intrinsic to the nature of the JUnit dataset. Furthermore, as detailed in chapter 4 and chapter 5, the datasets used for experiments are relatively small, three of them containing fewer than 950 samples. They exhibit significant class imbalance in terms of category distribution. This is also apparent from the bar plots Figure 5.1 - Figure 5.4. As a result, it is expected that JUnit, the smallest dataset with 252 samples, performs the worst. We suggest that, in addition to class imbalance, three other factors contribute to this outcome.

First, release log messages often include a link to a Jira issue. This pattern appears in datasets such as Project Lombok and Apache Commons IO. However, within the JUnit dataset, these messages are frequently brief and unclear. Without consulting the linked issue, they are sometimes hard to categorise, thus lowering the ML model metric scores.

The second factor affecting model performance on the JUnit dataset is the length of the release log text in the ***Changes*** column. Compared to Log4j, Apache Commons IO, and Project Lombok, the JUnit messages are the shortest, with the highest distribution featuring around 10 words, as visible from Figure 5.7. Contrarily, Project Lombok has 15 and 20 words, demonstrated in Figure 5.8. As a result, NLP and ML models face difficulties due to the limited information available for training and distinguishing between categories.

The last factor affecting the ML model's performance on the JUnit dataset is the infrequent use of keywords. In it, fewer messages contain keywords that directly indicate possible categorisation. For example, ***Bug fix*** is a particular phrase which, in almost all cases, signifies a bug-fixing release for known bugs. Similarly, the word ***Add*** prompts the possibility of extending the functionality of a class or method, for instance.

Overall, in Figure 5.9 - Figure 5.12, the performance of the classifiers is summarised for all datasets. The remaining classifiers exhibit more balanced performance scores. Nevertheless, a notable difference is evident between the macro and weighted average results in the classification report. Consider Table 5.4. The macro average recall, which measures the proportion of actual positive instances correctly identified by the model, is 0.523 higher for the weighted average compared to the macro average. This does not necessarily indicate that our model is poorly performing. However, the reason for this score is that the macro average is stricter. Meaning, that it lowers the result due to the subpar model performance on categories with a smaller sample size, such as ***Support for external development tool*** and ***Update dependency***. In the weighted average, emphasis is more on classes with higher sample sizes. Based on this information, it can be inferred that the models tend to perform better, with higher macro average values, when the sample size increases. To support this claim, we combined the aforementioned datasets into a single one and present the best-performing classifiers in Table 5.6 and Table 5.5. The results demonstrate an increase in efficiency.

Category Performance

After discussing the performance of the classifiers on the datasets, it is interesting to investigate how individual categories perform. Based on the confusion matrices in Figure 5.9 - Figure 5.12 and the classification reports such as Table 5.1, categories with consistently larger sample sizes, such as ***Bug fix*** and ***Add support for a new feature***, exhibit stable performance across evaluations. Their recall, precision and F1-score values are above 60% in most cases, except for the JUnit dataset, which was already discussed. Overall, beyond the sample size argument, another factor relates to the structured way in which release log messages are composed for the two categories. This further contributes to achieving higher scores.

Moving to other interesting categories, these include ***Update dependency*** and ***Documentation***. Except for JUnit and cases when a category is missing or has only one sample, both categories are classified with high scores, above 70%, for all evaluation metrics by the models. We argue that this is due to the nature of the release log messages that primarily apply to documentation updates or dependency changes. In both cases, the messages contain the relevant keyword, allowing the model to accurately classify the category of change. However, the JUnit dataset has release log messages such as *Parameterized, again [@orfjackal, fixing gh-285]*, that should be classified as ***Documentation***. Nevertheless, this is evident only after consulting the issue page linked to the release log message.

Finally, considering the worst-detectable categories, those can be classified into two variants. On the one hand, there are the underrepresented. For example, categories such as ***Performance improvement***, ***Support for external development tool***, and ***Quality Assurance*** have fewer than five test samples, implying only about 25 training samples. This makes them challenging for accurate classification by ML models. On the other hand, with categories such as ***Code redesign*** and ***Maintainability improvement***, the situation is different. They have higher representation, but score lower, compared to ***Bug fix*** and ***Add support for a new feature***. The reason is in the essence of the change. Both ***Code redesign*** and ***Maintainability improvement*** cover different subtypes of changes. Concerning the former, redesign, for instance, could be *change of implementation*, *changing access modifier* or *moving a file to a different package*. Further, ***Maintainability improvement*** is related to *simplification* and *refactoring*. The nature of these categories is inherently more complex for classification, as they lack distinctive keywords or patterns that the ML model can reliably learn from, further hindering its performance.

Classification Model Performance

The results for categorising release log messages showcase that ML models such as RFC and VC perform better compared to the rest. Moreover, as shown in the tables in chapter 5, four out of six present results related to the Random Forest Classifier (RFC), as it was the highest-performing model among those evaluated. The primary reason is the RFC's robustness in handling imbalanced datasets while delivering strong performance. Similarly, VC, which is the second in terms of performance, includes RFC as part of its three estimators.

5.3 Functionality and Architecture

5.3.1 Release Notes & Codebase Analysis

Before delving into the specific and interesting results, we explore the codebases of JUnit, Apache Commons IO and Project Lombok, intending to analyse common features they exhibit and share important remarks for them. Overall, from the categories listed in Table 4.3, only two are expected to influence the functionality of the API, namely, *Add support for a new feature* and *Deprecate functionality*, with both performing contrary actions. Regarding the impact on the architecture, all changes that alter the codebase may influence the architecture. These categories include *Code redesign*, and *Maintainability improvement*, as they encompass actions such as *Change implementation*, *Change access modifier*, *Move file to different package*, *Code simplification* and *Code*

refactoring. Thus, most of them alter the architecture of the code or package-based layer. The remaining categories of changes are expected not to influence the API's functionality or architecture. Nevertheless, this is something that we are empirically investigating.

The tools utilised for this research question were outlined in chapter 4. Overall, most of them worked correctly and produced the expected outputs for the majority of codebase versions. However, the arcana tool was unable to analyse the generated knowledge graph for Project Lombok versions 0.11.8 to 1.18.28 due to parsing exceptions. Apart from this limitation, arcana successfully processed the knowledge graphs for the remaining versions of Project Lombok, as well as JUnit and Apache Commons IO, generating inputs that were subsequently visualised using bubbleTea 2.0.

5.3.2 Results

In this subsection, all results related to RQ_2 are shared in the form of diagrams and tables. Class-level architecture, dependency model, release log messages and codebase statistics are explored to understand the evolutionary patterns. Each table and figure is discussed within the section corresponding to its respective codebase. From all available versions per API, the selected ones are those that exhibit changes affecting functionality and architectural structure.

JUnit

The first codebase examined is JUnit. Tables and diagrams for versions 4.6 and 4.11, alongside release log messages from version 4.9, are provided to analyse changes in functionality and architecture. First, in Table 5.7, some general statistics related to the codebase are displayed. Moving to the visualisation of the code organisation, part of it is presented. We showcase only the most interesting properties. The reason for this is due to the complexity and number of classes present, which impede the visualisation.

Table 5.7: Codebase statistics for selected JUnit versions

Version	Lines of Java Code	Number of Packages	Number of Classes	Number of Methods
4.6	8,620	25	179	851
4.11	11,824	28	217	1,097
4.12	15,274	30	276	1,338
4.13	17,488	32	332	1,488

The findings for the JUnit codebase, as measured through the MetricsReloaded tool, are summarised in Table 5.7. The results suggest that JUnit exhibits consistent growth in all measured aspects with each successive version.

Figure 5.13 visualises a segment of the class structure for JUnit version 4.6 codebase. From the class diagram, it is noticeable that classes are predominantly utilised in inheritance and the implementation of interfaces to achieve reusability. To avoid possible confusion, it is important to specify the meaning of the arrow colours. The purple indicates class inheritance achieved via the *extends* keyword in Java. The green arrow signifies the implementation of interfaces, achieved via the *implements* Java keyword.

Figure 5.14 represents the class diagram for JUnit after five *MINOR* versions starting from 4.6. The current structure is a continuation of version 4.6. The behaviour and functionality of existing classes are extended, such as the *Statement*.

Figure 5.15 presents a segment of the JUnit dependency model, illustrating its package-level dependencies. One noticeable characteristic of JUnit's package structure is the imbalanced distribution of classes. Some packages contain up to ten times more classes than others. This unevenness is also reflected in the dependency structure, where few packages exhibit significantly higher incoming and outgoing dependencies compared to the rest.

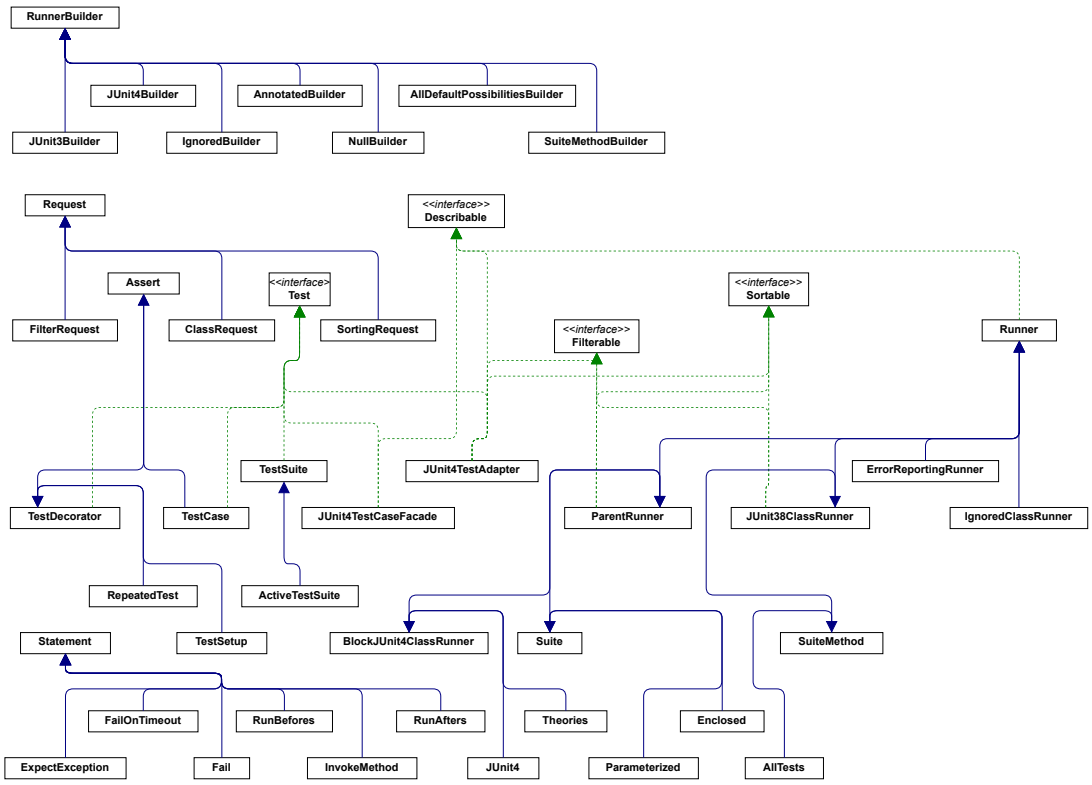


Figure 5.13: Part of the class diagram for JUnit version 4.6

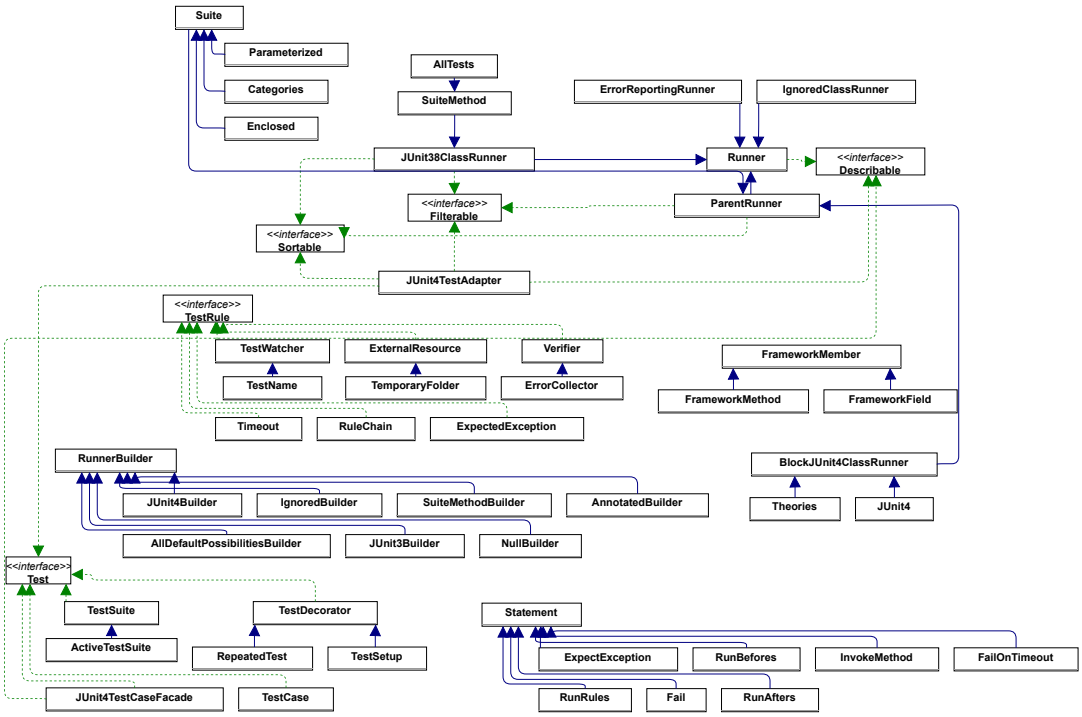


Figure 5.14: Part of the class diagram for JUnit version 4.11

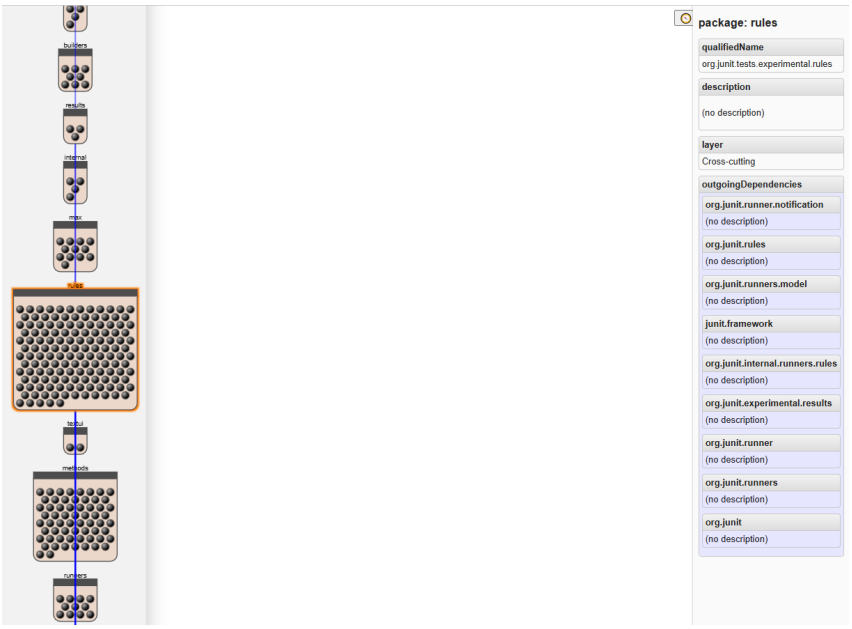


Figure 5.15: Package visualisation with dependencies for JUnit version 4.11

Table 5.8: Sample release log messages highlighting key updates in JUnit version 4.9

Version	Release	Changes	General Category
4.9	MINOR	ClassRule	Add support for a new feature
4.9	MINOR	Fields annotated with @Rule or @ClassRule should be of type TestRule	Code redesign
4.9	MINOR	-	Deprecate functionality
4.9	MINOR	Maven bundles and license updates	Code redesign
4.9	MINOR	The Common Public License included in source repository	Code redesign
4.9	MINOR	github#98: assumeTrue() does not work with expected exceptions	Bug fix
4.9	MINOR	github#74: Categories + Parameterized	Bug fix
4.9	MINOR	Removed folder "experimental-use-of-antunit", replaced by bash script	Maintainability improvement
4.9	MINOR	Various Javadoc fixes	Documentation
4.9	MINOR	Made MultipleFailureException public	Code redesign
4.9	MINOR	github#240: Add "test" target to build.xml	Maintainability improvement
4.9	MINOR	github#247: Give InitializationError a useful message	Bug fix

Table 5.8 presents a subset of the release log messages for JUnit version 4.9. The entries in the *Changes* column are often unclear and difficult to interpret. This poses a challenge when attempting to understand and categorise the messages based on their content. The issue is particularly evident for entries that reference GitHub pull requests, as these typically provide minimal context or explanation.

Apache Commons IO

Apache Commons IO provides a codebase for each of its 22 releases, in contrast to JUnit. Since the focus is on the changes impacting predominantly the architecture and functionality, ten versions

are selected and displayed in Table 5.9. Statistical information, analogous to that presented in the JUnit subsection, is provided. Moving to the diagrams of the code organisation, only part of the visualisations and versions are presented. This is similar to JUnit, where the complexity and size of each codebase version are the main factors.

Table 5.9: Codebase statistics for selected Apache Commons IO versions

Version	Source Code Lines	Number of Packages	Number of Classes	Number of Methods
1.1	7,785	4	41	386
1.3	11,563	4	58	549
1.4	14,326	5	76	655
2.0	21,165	6	104	968
2.2	22,798	6	107	1,046
2.5	26,018	7	122	1,186
2.7	31,949	10	173	1,541
2.8	32,956	10	179	1,603
2.9	36,316	11	197	1,822
2.12	46,447	14	290	2,563

The codebase statistics for selected versions of Apache Commons IO are summarised in Table 5.9. The results reveal that with each increment in the version, the codebase exhibits additional functionality and behaviour.

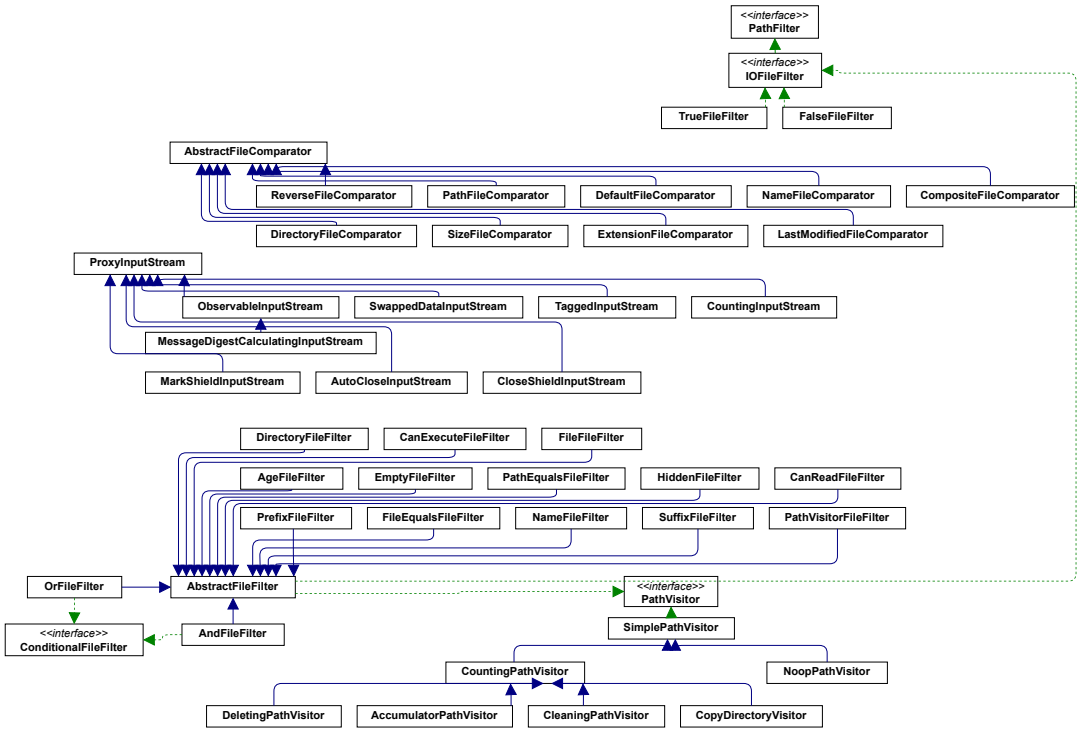


Figure 5.16: Part of the class diagram for Apache Commons IO version 2.9

Figure 5.16 displays a segment of the class structure from the Apache Commons IO version 2.9 codebase. Overall, the structure and organisation of the classes are again concentrated on inheritance. Key classes, such as *ProxyInputStream*, *AbstractFileFilter*, *PathFilter*, and *PathVisitor*, form the foundation for core behaviour and interfaces. These serve as base components, from

which child classes inherit and extend functionality to address specific requirements.

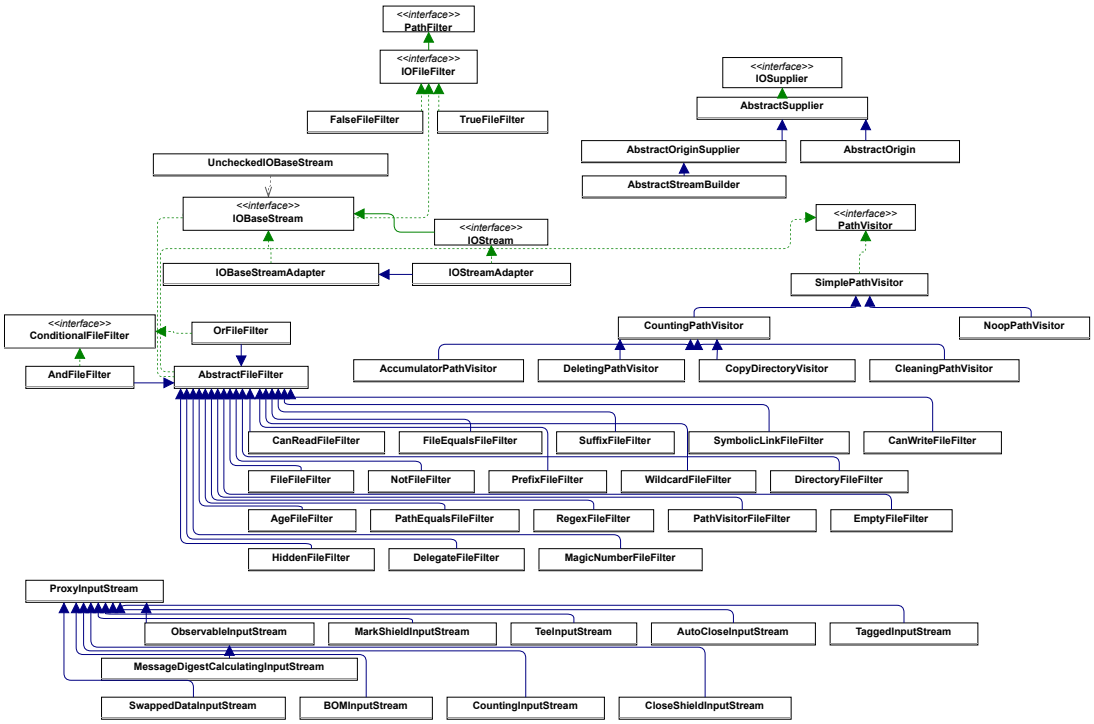


Figure 5.17: Part of the class diagram for Apache Commons IO version 2.12

Figure 5.17 shows part of the class diagram of Apache Commons IO three *MINOR* versions after Figure 5.16. Overall, the visualisation depicts that additional behaviour and functionality are added. Specifically, *ProxyInputStream* is extended with two classes and *AbstractFileFilter* with a few more. Except that the structure and organisation remain the same without noticeable reorganisations.
Figure 5.18 presents a share of the Apache Commons IO dependency model. In contrast to JUnit, the dependencies in Apache Commons IO exhibit a more balanced distribution, with packages containing a relatively similar number of classes. This enables better distribution of dependencies and responsibilities.

Table 5.10: Example of shorter release messages for Apache Commons IO version 1.1

Version	**Release**	**Changes**	**General Category**
1.1	MINOR	FilenameUtils - new class	Add support for a new feature
1.1	MINOR	FileSystemUtils - new class	Add support for a new feature
1.1	MINOR	IOUtils - new public constants	Add support for a new feature
1.1	MINOR	IOUtils - toCharArray(InputStream) [28979]	Add support for a new feature
1.1	MINOR	IOUtils - readLines(InputStream)	Add support for a new feature
1.1	MINOR	zIOUtils - toInputStream(String)	Add support for a new feature

The entries from Table 5.10 show examples of briefer release log messages. However, they do not hinder the process of understanding and categorising the change. This is in contrast to the properties showcased in the JUnit messages.

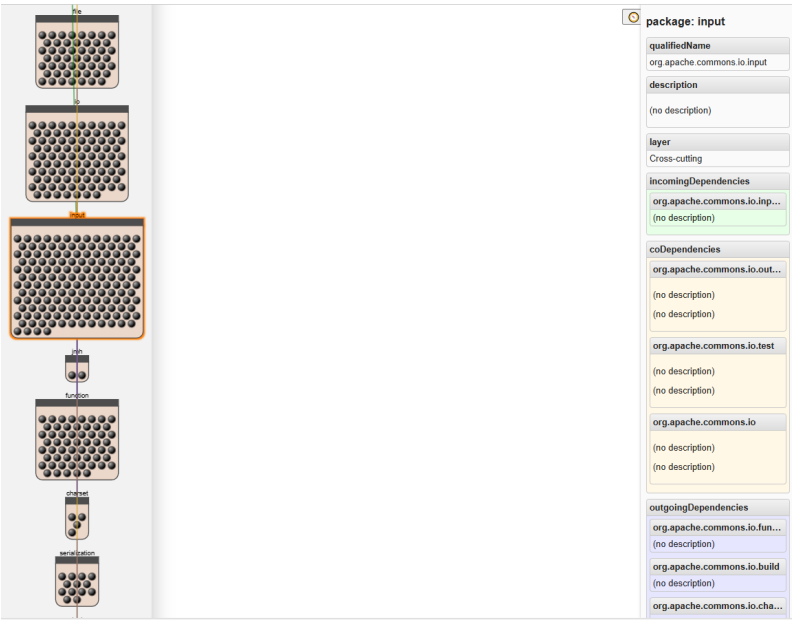


Figure 5.18: Package visualisation with dependencies for Apache Commons IO version 2.12

Table 5.11: Example of more elaborate release messages for Apache Commons IO version 1.4

Version	Release	Changes	General Category
1.4	MINOR	Add new Tee input stream implementation	Add support for a new feature
1.4	MINOR	Add new File Writer implementation that accepts an encoding	Add support for a new feature
1.4	MINOR	Add support for temporary files	Add support for a new feature
1.4	MINOR	Add a new write(InputStream) method [IO-152]	Add support for a new feature
1.4	MINOR	New Closed Input/Output stream implementations [IO-122]	Add support for a new feature
1.4	MINOR	Add Singleton Constants to several stream classes [IO-143]	Add support for a new feature
1.4	MINOR	Add facility to specify case sensitivity on prefix matching	Add support for a new feature
1.4	MINOR	Make IOFileFilter implementations Serializable [IO-131]	Code redesign
1.4	MINOR	Improve IOFileFilter toString() methods [IO-120]	Maintainability improvement
1.4	MINOR	Make fields final so classes are immutable/-threadsafe [IO-133]	Maintainability improvement
1.4	MINOR	Add a compare method to IOCase [IO-144]	Add support for a new feature
1.4	MINOR	Add a package of java.util.Comparator implementations for files [IO-145]	Add support for a new feature

Table 5.11 shows a higher quality of release log messages from version 1.4 of Apache Commons IO. Here, the messages are more elaborate and still easy to understand and categorise. Additionally, each entry includes a link to the corresponding pull request, allowing the reader to access supplementary information.

Project Lombok

Project Lombok has the highest number of releases among all API codebases, namely 57. Due to this, the focus is again predominantly on versions which cover architecture or functionality changes. Table 5.12 lists the general statistics codebase that should indicate the overall size of the codebase. Furthermore, it is important to specify that the statistics are obtained from the ***core*** folder of the codebase. Otherwise, the repository space is huge and unfeasible to analyse, with the arcana tool requiring one to two days to accomplish this.

Table 5.12: Codebase statistics for selected Project Lombok versions

Version	Source Code Lines	Number of Packages	Number of Classes	Number of Methods
0.9.3	11,400	10	114	639
0.10.0	16,129	15	175	945
1.16.6	29,621	19	377	1,646
1.16.8	30,362	19	386	1,693
1.16.22	32,577	20	423	1,875
1.18.10	37,580	21	472	2,126
1.18.12	38,782	21	479	2,224
1.18.16	41,195	22	500	2,313

The codebase statistics for selected versions of Project Lombok are summarised in Table 5.12. The results indicate that, similarly to the previous codebases, with each increment in the version, the codebase adds new functionality and behaviour.

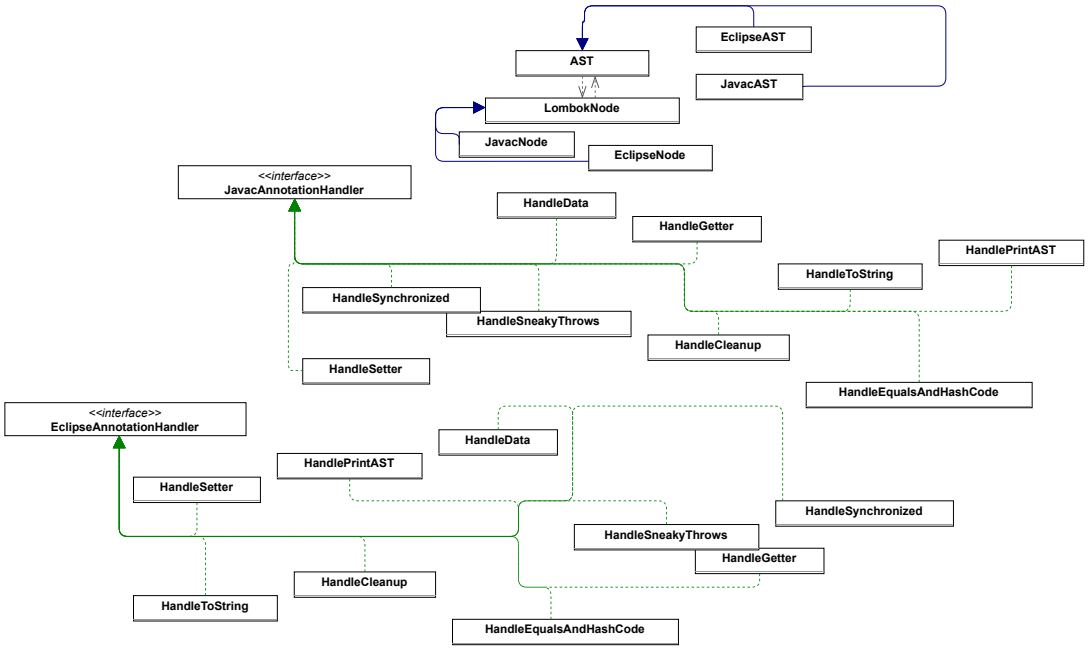


Figure 5.19: Part of the class diagram for Project Lombok version 0.9.3

The class-level architecture of Project Lombok features numerous important interfaces and classes, as depicted in Figure 5.19. Those are *JavacAnnotationHandler* and *AST*. They serve as a foundational layer for adding further implementation to the codebase. Overall, Project Lombok depicts a structured and organised codebase, suitable for evolution.

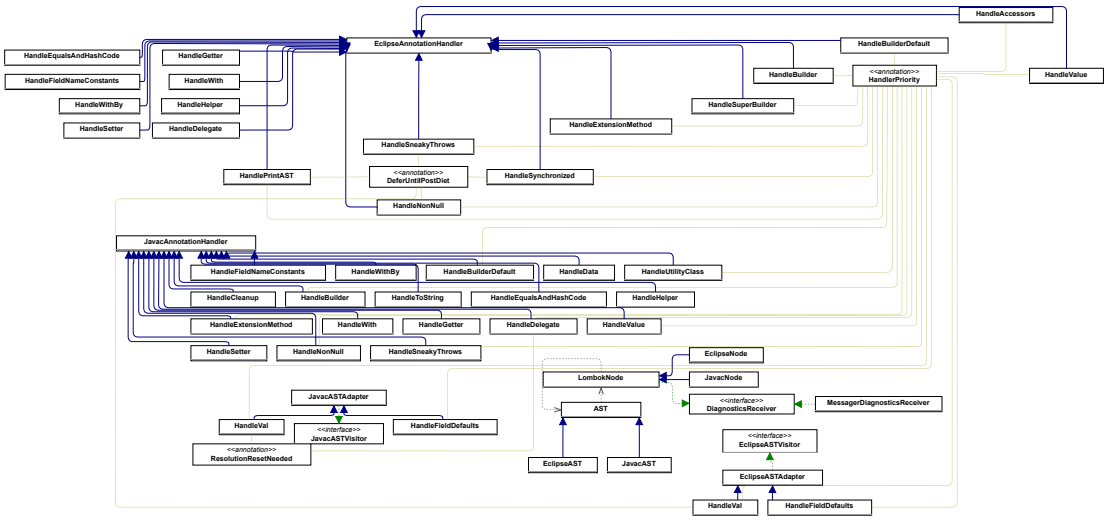


Figure 5.20: Part of the class diagram for Project Lombok version 1.18.16

Figure 5.20 depicts the class diagram of Project Lombok after several *PATCH*, *MINOR* and *MAJOR* version increments. The foundational classes identified in version 0.9.3 are further extended in subsequent versions, with the addition of other significant classes such as *EclipseAnnotationHandler*. Generally, the evolution process appears to be structured and organised, with no substantial alterations to the existing foundational codebase. The principle of being open to extension but closed to modification has been applied.

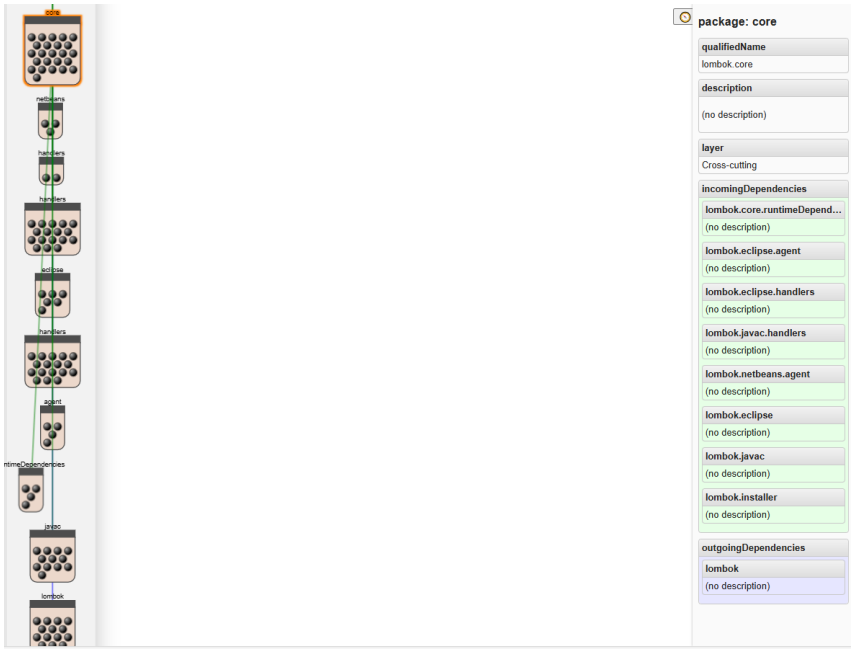


Figure 5.21: Package visualisation with dependencies for Project Lombok version 0.9.3

Among all dependency models, Project Lombok demonstrates the most balanced approach. From Figure 5.21 we can identify that packages are small, with a similar number of classes.

Table 5.13: Example release messages for Project Lombok from version 0.9.3

Version	Release	Changes	General Category
0.9.3	PATCH	FEATURE: Adding @Getter or @Setter to a class is now legal and is like adding those annotations to every non-static field in it. Issue #202	Add support for a new feature
0.9.3	PATCH	FEATURE: Three new annotations, @NoArgsConstructor, @RequiredArgsConstructor and @AllArgsConstructor have been added. These split off @Data's ability to generate constructors, and also allow you to finetune what kind of constructor you want. In addition, by using these annotations, you can force generation of constructors even if you have your own. Issue #152	Add support for a new feature
0.9.3	PATCH	FEATURE: generated toString, equals and hashCode methods will now use this.getX() and other.getX() instead of this.x and other.x if a suitable getter is available. This behaviour is useful for proxied classes, such as the POJOs that hibernate makes. Usage of the getters can be suppressed with @ToString/@EqualsAndHashCode(doNotUseGetters = true). Issue #183	Code redesign
0.9.3	PATCH	PLATFORMS: Lombok should now run in stand-alone ecj (Eclipse Compiler for Java). This isn't just useful for the few souls actually using this compiler day to day, but various eclipse build tools such as the RCP builder run ecj internally as well. Issue #145	Bug fix
0.9.3	PATCH	BUGFIX: Eclipse: @Data and other annotations now don't throw errors when you include fields with bounded wildcard generics, such as List¡? extends Number¿. Issue #157	Bug fix
0.9.3	PATCH	BUGFIX: When @Getter or @Setter is applied to a multiple field declaration, such as @Getter int x, y;, the annotation now applies to all fields, not just the first. Issue #127	Bug fix
0.9.3	PATCH	BUILD: dependencies are now fetched automatically via ivy, and most dependencies now include sources by default, which is particularly handy for those working on the lombok sources themselves.	Maintainability improvement

Table 5.13 features the release log messages for Project Lombok version 0.9.3. From all codebases, Project Lombok has the most elaborate and formatted variant of messages. Particularly, each entry provides enough details for the change, while also adding a link to a pull request. Before the start of the release log text, a word in capital letters represents what aspects of the codebase have been changed. This can significantly aid the reader in understanding what has been performed.

5.3.3 Discussion

In the previous subsection, the results for JUnit, Apache Commons IO, and Project Lombok were displayed in the form of class diagrams, software metrics and excerpts from release log messages. Following this information, in the current subsection, we explore the implications of the results. Initially, we focus on each codebase to investigate the general trends. For a more structured approach, we analyse each codebase based on three folds.

The analysis first examines the distribution of categories in release log messages to reveal a tendency in codebase alterations. The second fold involves analysing the messages from the respective versions of the dataset and connecting findings with results from RQ_1. In this manner, it can be investigated from the style of writing and contents whether changes impacting the architecture and functionality are presented in the release log messages. Finally, the diagrams and codebase statistics are analysed to assess the architectural impact of evolution. Patterns of change are identified by examining selected versions.

Interpretation of Results

JUnit From the selected APIs, JUnit has the smallest number of changes and releases covered. Most of the changes are dominated by the categories *Bug fix*, *Add support for a new feature* and

Code redesign. The latter two indicate an influence on the functionality and architecture of the system.

Part of the release log messages for JUnit are difficult to understand due to their content and writing style. Consider Table 5.8, the messages are brief and do not necessarily explain what exactly has been changed. Therefore, it is sometimes challenging to infer what the alternations influence. This phenomenon was already experienced and explained in the JUnit part for RQ_1. There, we shared that the messages are challenging to categorise due to their diverse and unsystematic manner of writing. Moreover, for some pull requests, links to a version control repository such as GitHub are provided. Therefore, to understand what has changed, the reader should visit the links. The GitHub pull requests have replaced the previous style of writing since version 4.12. Starting from it, virtually any release log message is a link to one. Before that, this approach had significantly less usage. One possible reason is that the JUnit team began using GitHub's release functionality starting with version 4.11. Consequently, for subsequent versions, they decided to structure their messages in a manner where each release log message is attributed to a specific pull request.

Moving to the code organisation and evolution, an interesting characteristic is the package structure. From Table 5.7, we can deduce that the maintainers and developers separated the code into multiple packages. This started with 25 in version 4.6 and went up to 32 in version 4.13. Furthermore, the same table shows that with each version, the codebase evolved to include more packages, classes, and methods. As a result, additional behaviour has been added. The package organisation shows an attempt to organise the code in a logically coherent manner. Nevertheless, an interesting observation is the difference in the number of classes per packet. Figure 5.15 illustrates part of the packets and their corresponding classes. There are several packages, such as *org.junit.tests.experimental.rules*, and *org.junit.tests.running.methods* that have ten times more classes, compared to others such as *org.junit.runners* and *org.junit.tests.experimental.max*. However, some packages can be combined since they exhibit the same structure. For instance, *org.junit.samples.money* and *junit.samples.money* have a single class for the whole package, which can just be transferred to the latter one. Regarding class diagram organisation, Figure 5.13 and Figure 5.14 depict two codebase versions. Since the gap between them is five *MINOR* versions, there are noticeable differences in the class-level architecture. Moreover, the alternations between both versions are depicted in the release log messages. For instance, in 4.7, the classes ***TestName.java*** and ***TemporaryFolder.java*** are added to the codebase, and they are reflected in 4.11. The architecture structure in the two versions is similar, with fundamental classes such as ***Statement.java*** and ***RunnerBuilder.java*** being present in both. Furthermore, the JUnit codebase is utilising the *Strategy* design pattern in which can be observed by the implementation of the ***Sortable.java*** and ***Filterable.java*** interfaces by the various Java *Runners* classes. Overall, from the architecture and codebase, we can infer that changes that impact the architecture and functionality are represented in the release log.

Apache Commons IO In contrast to JUnit, Apache Commons IO has the codebase for its first iteration available. This allows for more precise examination of its evolution over time. Similarly to JUnit, the most popular categories of change are *Bug fix* and *Add support for a new feature*. However, *Update dependency* and *Maintainability improvement* are the next ones as opposed to *Code redesign*. In general, of the categories listed above, only *Add support for a new feature* and *Maintainability improvement* are likely to affect the system's functionality and architecture.

The release log messages in Apache Commons IO exhibit a similar structure. This is in contrast to JUnit, for instance. Notably, all Apache Commons IO messages within a certain version exhibit a consistent writing style. In the first version of the project, the release log messages are predominantly shorter. Table 5.10 shows part of them. However, they still indicate what changes have been performed. As the project evolves, an increasing number of release messages are linked to a specific issue. In this case, Jira is selected as the tracking tool. An interesting observation from Table 5.11 is that the release messages do not deteriorate in quality. It is clear without visiting the link to the Jira issue what has been performed, thus making it easy to categorise it.

The code organisation and evolution of Apache Commons are different from JUnit. One noticeable difference is the less fine-grained usage of packages. As displayed in Table 5.9, version 2.12 has 14 packages, compared to 32 in version 4.13 of JUnit. Furthermore, the Apache codebase is significantly larger than its counterpart, with 46,447 lines of source code compared to 17,488. Another interesting point is the ratio between classes and methods. For all versions, Apache has a higher average number of methods per class compared to JUnit. This possibly indicates that it contains more behaviour inside one class. Regarding the evolution of the Apache codebase, Table 5.9 shows a gradual increase in the size and functionalities. The dependency of Apache Commons IO exhibits properties similar to those of JUnit. Some packages, as visible from Figure 5.18, such as *org.apache.commons.io* and *org.apache.commons.io.input*, have a considerably higher number of classes compared to other packages. Nevertheless, the code is better spread, since the smallest number of classes present in a package is two. This indicates a more balanced separation of concerns and a less fine-grained approach in organising the codebase. The evolution of Apache Commons IO is connected with the addition of more behaviour and functionalities. From the class diagrams in Figure 5.16 and Figure 5.17, it is evident that the class-level architecture of the system remains largely unchanged. Similar to the JUnit codebase, the *Strategy* design pattern is employed to structure parts of the code, where the interface and abstract class ***IOFileFilter.java*** and ***ProxyInputStream.java***, respectively, declare the behaviour. Additionally, we can also spot the *Visitor* design pattern being utilised in ***PathVisitor.java***. Although the difference between the two Apache Commons IO versions spans three *MINOR* releases, the overall architecture remains consistent, despite the addition of new functionality. Generally, based on our observations, the release log messages clearly describe the changes made, including those that affect the system's functionality and architecture.

Project Lombok Project Lombok exhibits interesting behaviour regarding the categories of changes that are performed on the codebase. Regarding the most common modifications, the majority are *Bug fix* (56.14%) and *Add support for a new feature* (24.55%), followed by *Code redesign* and *Maintainability improvement*. In general, this indicates that the majority of the work has been focused on correcting issues rather than introducing new functionality.

Compared to Apache Commons IO and JUnit, Project Lombok has the most frequent releases, 57. This possibly indicates that the maintainers prefer to evolve the codebase in small increments and emphasise PATCH releases. Moving to the release log messages, they have a standardised style of writing. Similarly to JUnit, most of them are links to a GitHub pull request issue. Nevertheless, the structure of the messages is clearer. Consider Table 5.13, each one has a keyword at the beginning, indicating the type of change, followed by a comprehensive explanation regarding what has been changed. The approach was first implemented in version 0.9.3. In earlier versions, release logs focused on describing the changes made, often without using specific keywords. Although, as expected that each keyword, such as *ENHANCEMENT*, *FEATURE*, or *BUGFIX*, signifies a specific category of change, this is not always the case. Nevertheless, for the most common categories of change, such as *Bug fix* and *Add support for a new feature*, the consistency is present.

The code organisation of Project Lombok is distinct from Apache Commons IO and JUnit. First and foremost, we perform the analysis for Project Lombok solely on the ***src/core*** folder, as specified in the Results section. Starting with the package structure organisation, as shown in Table 5.12, it commences with ten packages in version 0.9.3 and by 1.18.16 it has 22. This suggests a more fine-grained approach in the separation of functionalities and responsibilities, similar to JUnit. Moving back to the package and class structure, the bubbleTea 2.0 tool visualisation Figure 5.21 illustrates balanced usage of packages, indicating good separation of responsibilities. Overall, compared to JUnit and Apache Commons IO, Project Lombok demonstrates a more even distribution of responsibilities. While *lombok.core* is heavily utilised and interacts with multiple other packages, the remaining exhibit relatively even usage patterns. Concerning the system architecture, the most notable change between versions is the inclusion of additional classes and interfaces. The foundational architecture shown in Figure 5.19 for version 0.9.3 is extended in later versions with additional behaviour. From both class diagrams, we assure that the architecture

follows a clear structure with evident separation of concerns. With respect to design patterns, the *Strategy* is present with many different classes extending the ***EclipseAnnotationHandler.java*** and ***JavacAnnotationHandler.java***. Although much has changed between versions 0.9.3 and 1.18.16, the core functionality and architecture remain intact and have been extended rather than altered. Moving to the release log messages, those that impact the architecture and functionality, are predominantly clear. For instance, in version 1.18.16, a maintainer signifies that the performed change is a breaking change. This informs the community of the possible consequences of implementing it, which might break previous compatibility. Regarding the architecture, in most cases, it is not clear when new classes or methods are added to the codebase. Instead, the maintainer is focusing on what behaviour is achieved when adding the new methods or class.

General Observations

From the threefold investigation on APIs architecture and functionalities, we can recapitulate several points. For the three codebases, most changes are related to ***Bug fix*** and ***Add support for a new feature***. The latter specifically suggests that an alteration to the functionality and architecture is expected. Such changes are communicated via the release log messages. However, for JUnit, it is not always clear what is modified without referencing the corresponding pull request. Regarding the architecture and functionalities, we explored and established that all codebases have a structured class-level architecture. Furthermore, the *Strategy* design pattern is present in all three codebases, indicating that all of them value the flexibility it provides. The *Visitor* pattern is also present, but not in all codebases. Over time, the architecture and functionality have evolved primarily through extensions of existing behaviour and bug fixes. Therefore, new behaviour is predominantly added while deprecation is rarer.

5.4 Research Questions Answers

After presenting the results and discussing their implications, in this section, we focus on using the findings to answer the research questions.

RQ_1 : ***To what extent can an automated machine learning technique analyse and categorise the causes of changes in Java APIs?***

Following the results and discussion, we can conclude that ML techniques can effectively categorise changes. They especially benefit when a larger dataset, containing more than 400 samples, for instance. Furthermore, if the data is balanced, performance is likely to improve, as more data provides better training resources for each class. Additionally, release-log messages must be detailed and structured. Having brief phrases or single-word entries hinders both the ML model's performance and humans' ability to categorise messages without referencing tools with additional information, such as Jira or GitHub. Even so, the model still achieved an accuracy of 70% or higher across most datasets. Finally, it can be suggested that the existing ML approach, with minor adjustments, is suitable for categorising changes in APIs beyond the Java ecosystem. The only requirement is that the respective dataset with changes has a representative sample size and balanced distribution of categories.

$RQ_{2.1}$: ***Are changes impacting the architecture and functionality represented in the release log?***

In the JUnit, Apache Commons IO and Project Lombok datasets and codebases, such changes are represented in the release log messages. In some cases, the changes are not directly stated in the release log messages but linked to a pull request or issue. The majority of release log messages adhere to a writing style established by the API maintainers. The exception is JUnit, which has an unsystematic manner of writing. Nevertheless, from all release log messages of APIs, we can infer the changes impacting functionality and architecture.

$RQ_{2.2}$: ***How does the architecture of an API alter during evolution?***

The architecture of an API evolves gradually over time. There is an increased number of classes and interfaces after each new version, which adds a layer of new functionality and abstraction. Design patterns (e.g. *Visitor* and *Strategy*) are present from the beginning of the codebase development and are extensively extended through the evolution. This aids in the organised evolution of the codebases. Depending on the initial structure of the codebase, some developers prefer a more fine-grained organisation that includes multiple packages, such as JUnit and Project Lombok. Others, as Apache Commons IO, adopt a less fine-grained approach.

RQ_2 : ***How do changes impact the architecture and functionalities of APIs?***

Changes to APIs influence both architecture and functionality. These modifications are generally reflected in release log messages, though the clarity and consistency vary. Architecturally, APIs evolve by adding classes, interfaces, and abstraction layers. Overall, more behaviour and functionality is added to APIs than deprecated. While some projects prefer fine-grained structures (e.g., JUnit, Lombok), others like Apache Commons IO utilise a less fine-grained structure. Furthermore, design patterns such as the Strategy and Visitor patterns are commonly adopted and extended during the evolution of the APIs.

Chapter 6

Threats to Validity

In this chapter, we discuss the various threats to validity arising from our research design. Specifically, we focus on threats across two key dimensions, namely internal and external. Internal validity refers to the confidence that the observed causal relationship is accurate and not distorted by confounding variables or external influences. On the other hand, external validity concerns how well the findings can be generalised beyond the specific context or dataset used in the research.

6.1 Internal Validity

One of the main threats to internal validity pertains to RQ_2. There, the argumentations are based on our interpretation of the results. We followed a structured approach centred on three investigative focuses regarding the codebases and datasets. Overall, the discussion and findings were based on empirical investigation, which may introduce potential bias or misinterpretation. Moreover, in terms of information, the main sources were the repositories of the codebases and links to pull requests. In some cases, earlier versions of the codebases could not be retrieved due to their unavailability. Therefore, processing them was limited to consulting solely the release log changes and an inability to focus on architectural or functionality aspects. By adopting a structured and organised approach, we aimed to minimise the risk of bias or misinterpretation. Another threat concerns how the datasets were constructed and categorised. Overall, the majority of this work was performed by Nikolova [2] in her study, and we verified all her categorisations of release log messages across the datasets. For Log4J and AppCompat, we constructed the datasets from scratch, involving them in several rounds of revisions and corrections. The primary objective was to maintain consistency in our categorisation approach, aligning it with the methodology employed by Nikolova. Finally, the selected evaluation metrics for RQ_1 are another threat to validity. We focus on metrics such as accuracy, precision, recall, F1-score and their scope, namely macro and weighted. Although these offer valuable insights into model performance, additional visualisation techniques, such as *Receiver Operating Characteristics Curve (ROC)* and *Area Under the Curve (AUC)* curves, can offer additional insights that enhance the current findings.

6.2 External Validity

The first and most important threat to external validity is related to the programming languages and APIs. The current research is focused on APIs from the Maven repository written in Java. Therefore, the results apply to Java and how such code is constructed and written. With respect to other programming languages, including Python, Rust, and C++, no experiments were performed. Nevertheless, especially for RQ_1, the selected methods should be applicable since the release log messages are investigated, not the code quality. Therefore, the practical approach for RQ_1 is language agnostic. Another threat to the external validity is related to the number and size of the datasets. The current study for RQ_1 presents the results for four datasets, with each release log

message assigned to one of the categories from Table 4.3. All of the investigated datasets feature imbalances in the categories. Overall, the minority classes are underrepresented in virtually all datasets. Nevertheless, we argue that for this research, the number of datasets and size are sufficient and do not influence the external validity.

Chapter 7

Conclusion & Future Work

This final chapter summarises the study by revisiting the main goal and research questions and highlighting the key findings. Furthermore, we specify our contributions to the research domain and provide an overview of the impact of our findings on the research area. Finally, we suggest some possible future directions that could extend our work and enable a deeper study and investigation of the domain.

7.1 Conclusions

In this study, we concentrated on the topic of evolutionary changes in APIs. The main goal was to *analyse the reasons for changes in APIs and their impact on the functionalities and architectures of the API*. To achieve this, we employ NLP and ML techniques, alongside an empirical investigation into the APIs' functionality and architecture. The results show that automatically categorising reasons for their changes is feasible via NLP and ML. Particularly, it can achieve accuracy above 70% for the majority of datasets, even for imbalanced ones. We argued that the results could be enhanced if the datasets with release log messages exhibited a more balanced distribution across the different categories. Another important point is the writing style of release log messages. In case they are descriptive and extensive, this aids the ML models in correctly classifying the changes. In contrast, brief messages with links to pull requests hinder the process. Moving to the impact on functionality and architecture, our results suggest that API changes do influence both aspects. APIs evolve architecturally through the introduction of new abstractions, such as classes and interfaces. Additionally, codebases exhibit varying levels of structural granularity across different projects. Throughout the evolution of codebases, well-known design patterns, such as the *Visitor* and *Strategy* patterns, are often adopted. Functionality is predominantly extended with each version increment, with only a few instances of deprecated features. Finally, we discovered that both changes impacting the architecture and functionality of APIs are depicted in release log messages. However, in some situations, the information is not directly available on the message, but after consulting a linked pull request.

With this study, we contribute to the domain of APIs by providing practical information regarding the causes of their changes and the respective effect on the externalisation of functionality and architecture. This has been a missing gap in research, and our research contributes to filling it. Overall, our findings described how APIs evolve and what types of alterations are required during this process. The insights from this study can help software professionals and researchers design and maintain more resilient systems by anticipating and managing API evolution. Additionally, our focus on release log messages emphasises the value of clear documentation in improving the understanding of evolutionary patterns and facilitating the categorisation of changes.

7.2 Future Work

Considering the future directions, the current thesis has various options for extension. One of them is to expand the analysis of release log messages to APIs beyond the Maven repository. In particular, widely utilised ecosystems such as Python (e.g. PyPI) and JavaScript (e.g. npm) represent promising candidates, as they have central role in shaping software development over the past decade. Another possible future direction is to employ different ML models for classifying release log messages. This has the potential to improve the performance of those models, possibly utilising ones from the TensorFlow library or employing Deep Learning (DL). Hyperparameter tuning associated with the RQ_1 experiment presents opportunities for further refinement and exploration. In our experiment, this is accomplished with a maximum of 120 candidates. Therefore, to identify better-performing hyperparameters, the range of parameter values can be expanded, combined with the selection of different ML models. Lastly, another future direction is connected to the impact of changes on the functionality and architecture of the APIs. It is worth investigating if the patterns observed in selected Maven libraries are similar to those in PyPI repositories. Alternatively, do the patterns resemble those observed in Maven libraries, or are there significant differences?

Bibliography

[1] S. A. Rukmono, "rsatrioadi/javapers: javapers 1.0," Jan. 2023. [Online]. Available: https://github.com/rsatrioadi/javapers?tab=readme-ov-file vii, 24, 25

[2] A. Nikolova, "How and Why Do Java Library APIs Evolve?" *Unknown*, 2023. ix, 1, 9, 10, 11, 12, 14, 15, 16, 18, 24, 52

[3] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang, "A Systematic Review of API Evolution Literature," *ACM Computing Surveys*, vol. 54, pp. 1–36, Oct. 2021. 1, 2, 4, 9, 12

[4] M. Meng, S. Steinhardt, and A. Schubert, "How developers use API documentation: an observation study," *Commun. Des. Q. Rev*, vol. 7, no. 2, pp. 40–49, Aug. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3358931.3358937 1

[5] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, Sep. 1980, conference Name: Proceedings of the IEEE. [Online]. Available: https://ieeexplore.ieee.org/document/1456074 1

[6] C. Kemerer and S. Slaughter, "An empirical approach to studying software evolution," *Software Engineering, IEEE Transactions on*, vol. 25, pp. 493–509, Aug. 1999. 1

[7] "The Most Popular Programming Languages - 1965/2024 - New Update -." [Online]. Available: https://statisticsanddata.org/data/most-popular-programming-languages-1965-2024/ 1, 25

[8] "Most used languages among software developers globally 2024." [Online]. Available: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/ 1

[9] D. Dig and R. Johnson, "How do APIs evolve? A story of refactoring," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.328. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.328 1, 4, 9, 10, 11, 12

[10] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of API-level refactorings during software evolution," in *Proceedings of the 33rd International Conference on Software Engineering*. Waikiki, Honolulu HI USA: ACM, May 2011, pp. 151–160. [Online]. Available: https://dl.acm.org/doi/10.1145/1985793.1985815 1, 9, 10

[11] L. Ochoa, T. Degueule, J.-R. Falleri, and J. Vinju, "Breaking bad? Semantic versioning and impact of breaking changes in Maven Central: An external and differentiated replication study," *Empirical Software Engineering*, vol. 27, no. 3, p. 61, May 2022. [Online]. Available: https://link.springer.com/10.1007/s10664-021-10052-y 1, 2, 4, 9, 10, 12

[12] "Maven Repository: Central." [Online]. Available: https://mvnrepository.com/repos/central 1, 15

[13] "Maven Central." [Online]. Available: https://central.sonatype.com 1

[14] N. Rozanski and E. Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*, 2nd ed. Addison-Wesley Professional, Oct. 2011. 2, 5, 11

[15] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Why and How Java Developers Break APIs," Jan. 2018, arXiv:1801.05198. [Online]. Available: http://arxiv.org/abs/1801.05198 2, 9, 11, 12

[16] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the Maven repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, Jul. 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121216300243 2, 9, 10, 12

[17] S. Serbout and C. Pautasso, "How Many Web APIs Evolve Following Semantic Versioning?" in *Web Engineering*, K. Stefanidis, K. Systä, M. Matera, S. Heil, H. Kondylakis, and E. Quintarelli, Eds. Cham: Springer Nature Switzerland, 2024, pp. 344–359. 4, 9, 10

[18] A. Nesbitt, "From ZeroVer to SemVer: A Comprehensive List of Versioning Schemes in Open Source," Jun. 2024. [Online]. Available: http://nesbitt.io/2024/06/24/from-zerover-to-semver-a-comprehensive-list-of-versioning-schemes-in-open-source.html 4

[19] "Semantic Versioning 2.0.0 | Semantic Versioning," 2013. [Online]. Available: https://semver.org/spec/v2.0.0.html 4

[20] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999. 5

[21] "ISO/IEC/IEEE 42010:2011." [Online]. Available: https://www.iso.org/standard/50508.html 5

[22] M.-C. Lee, "Software measurement and software metrics in software quality," *International Journal of software engineering and its application*, vol. 7, pp. 15–33, 01 2013. 5

[23] E. Bouwers, J. Visser, and A. Van Deursen, "Getting what you measure," *Communications of the ACM*, vol. 55, no. 7, pp. 54–59, Jul. 2012. [Online]. Available: https://dl.acm.org/doi/10.1145/2209249.2209266 6

[24] I. H. Sarker, "Machine Learning: Algorithms, Real-World Applications and Research Directions," *SN Computer Science*, vol. 2, no. 3, p. 160, Mar. 2021. [Online]. Available: https://doi.org/10.1007/s42979-021-00592-x 6

[25] D. Khurana and A. Koli, "(PDF) Natural language processing: state of the art, current trends and challenges," *ResearchGate*, Apr. 2025. [Online]. Available: https://www.researchgate.net/publication/319164243_Natural_language_processing_state_of_the_art_current_trends_and_challenges 7

[26] R. Koçi, X. Franch, P. Jovanovic, and A. Abelló, "Classification of Changes in API Evolution," in *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, Oct. 2019, pp. 243–249, iSSN: 2325-6362. [Online]. Available: https://ieeexplore.ieee.org/document/8944981 9, 12

[27] S. Sohan, C. Anslow, and F. Maurer, "A Case Study of Web API Evolution," in *2015 IEEE World Congress on Services*, Jun. 2015, pp. 245–252, iSSN: 2378-3818. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7196531 9, 10

[28] W. Li, F. Wu, C. Fu, and F. Zhou, "A Large-Scale Empirical Study on Semantic Versioning in Golang Ecosystem," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2023, pp. 1604–1614, iSSN: 2643-1572. [Online]. Available: https://ieeexplore.ieee.org/document/10298458 9, 11

[29] D. Pinckney, F. Cassano, A. Guha, and J. Bell, "A Large Scale Analysis of Semantic Versioning in NPM," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, May 2023, pp. 485–497, iSSN: 2574-3864. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10174013 9, 11

[30] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, and Y. Xiong, "How Do Python Framework APIs Evolve? An Exploratory Study," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2020, pp. 81–92, iSSN: 1534-5351. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9054800 9, 11

[31] S. A. Rukmono, M. R. V. Chaudron, and C. Jeffrey, "Layered BubbleTea Software Architecture Visualisation," in *Proceedings of the IEEE International Working Conference on Software Visualization (VISSOFT)*. IEEE Computer Society, Oct. 2024, pp. 122–126. [Online]. Available: https://www.computer.org/csdl/proceedings-article/vissoft/2024/284800a122/22MbVbLCVkA 9, 11, 12

[32] S. A. Rukmono, L. Ochoa, and M. Chaudron, "Deductive Software Architecture Recovery via Chain-of-thought Prompting," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. Lisbon Portugal: ACM, Apr. 2024, pp. 92–96. [Online]. Available: https://dl.acm.org/doi/10.1145/3639476.3639776 9, 12

[33] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012. 11

[34] R. Kakkenberg, S. A. Rukmono, M. Chaudron, W. Gerholt, M. Pinto, and C. R. De Oliveira, "Arvisan: an Interactive Tool for Visualisation and Analysis of Low-Code Architecture Landscapes," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*. Linz Austria: ACM, Sep. 2024, pp. 848–855. [Online]. Available: https://dl.acm.org/doi/10.1145/3652620.3688331 11

[35] D. Khyani, S. B S, N. Niveditha, D. M., and D. Y M, "An interpretation of lemmatization and stemming in natural language processing," *Shanghai Ligong Daxue Xuebao/Journal of University of Shanghai for Science and Technology*, vol. 22, pp. 350–357, 01 2021. 21

[36] M. Das, S. K, and P. J. A. Alphonse, "A Comparative Study on TF-IDF feature Weighting Method and its Analysis using Unstructured Dataset," Aug. 2023, arXiv:2308.04037 [cs]. [Online]. Available: http://arxiv.org/abs/2308.04037 21

[37] "RandomOverSampler — version 0.13.0." [Online]. Available: https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.RandomOverSampler.html 22

[38] "GridSearchCV." [Online]. Available: https://scikit-learn/stable/modules/generated/sklearn.model_selection.GridSearchCV.html 22

[39] "1.11. ensembles: Gradient boosting, random forests, bagging, voting, stacking." [Online]. Available: https://scikit-learn/stable/modules/ensemble.html 22, 23

[40] "1.4. support vector machines." [Online]. Available: https://scikit-learn/stable/modules/svm.html 23

[41] "f1_score," Apr. 2025. [Online]. Available: https://scikit-learn/stable/modules/generated/sklearn.metrics.f1_score.html 24

[42] B. Leijdekkers, "MetricsReloaded - IntelliJ IDEs Plugin | Marketplace," 2025. [Online]. Available: https://plugins.jetbrains.com/plugin/93-metricsreloaded 24

[43] S. A. Rukmono, "rsatrioadi/arcana," Feb. 2025, original-date: 2024-06-07T07:31:21Z. [Online]. Available: https://github.com/rsatrioadi/arcana 24, 25

[44] ——, "rsatrioadi/bubbletea-v2," Feb. 2025, original-date: 2024-11-28T16:47:39Z. [Online]. Available: https://github.com/rsatrioadi/bubbletea-v2 24

[45] "UML class diagrams | IntelliJ IDEA." [Online]. Available: https://www.jetbrains.com/help/idea/class-diagram.html 25

[46] "Top 26 Python Libraries for Data Science in 2024." [Online]. Available: https://www.datacamp.com/blog/top-python-libraries-for-data-science 25

Appendix A

Violin Plots

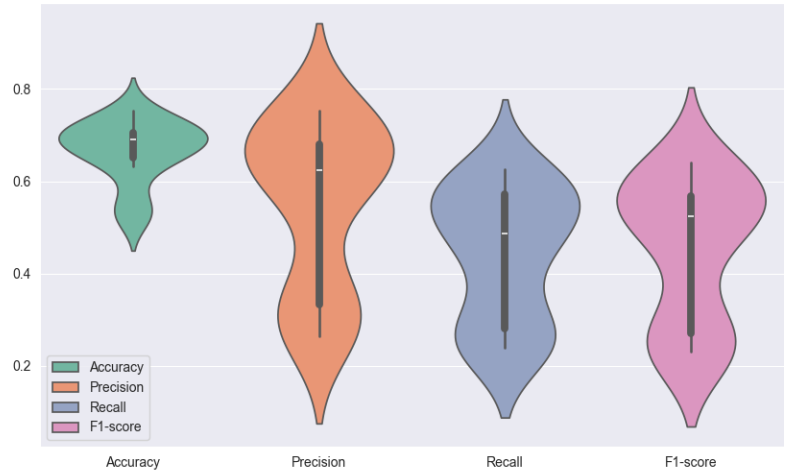


Figure A.1: Violin plot depicting the distribution per metric for the JUnit, Log4j, Project Lombok and Apache Commons IO datasets, utilising the four classifiers

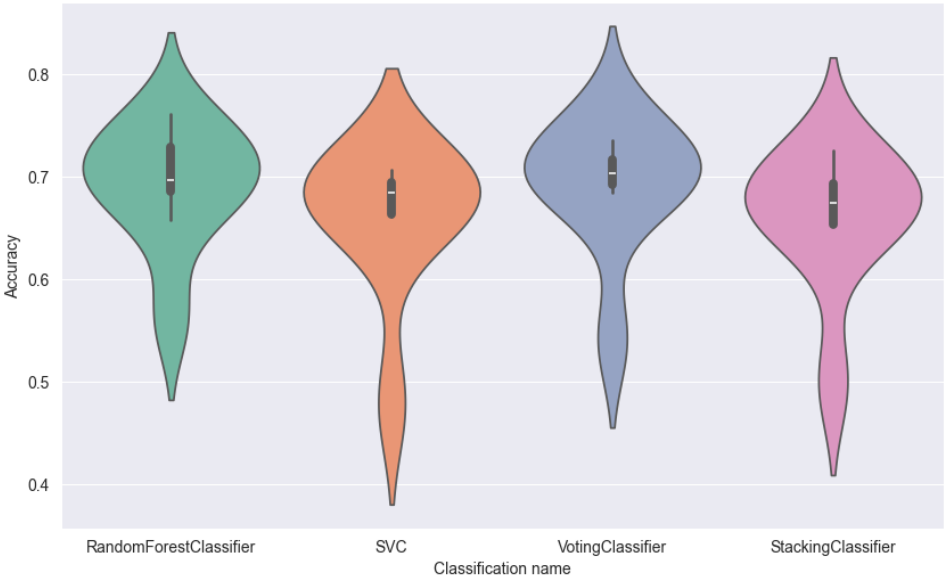


Figure A.2: Violin plot showing the accuracy distribution across classification models

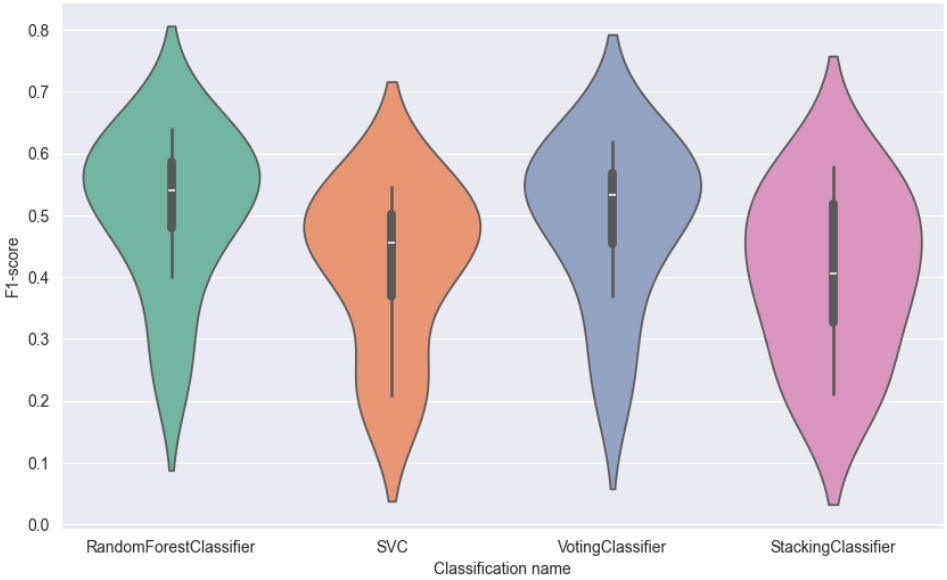


Figure A.3: Violin plot showing the F1-score distribution across classification models

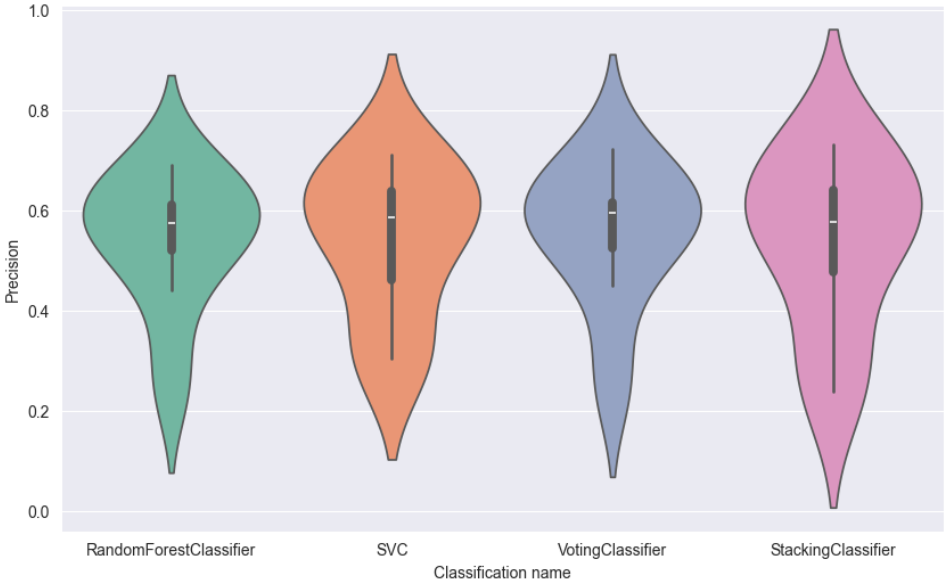


Figure A.4: Violin plot showing the precision distribution across classification models

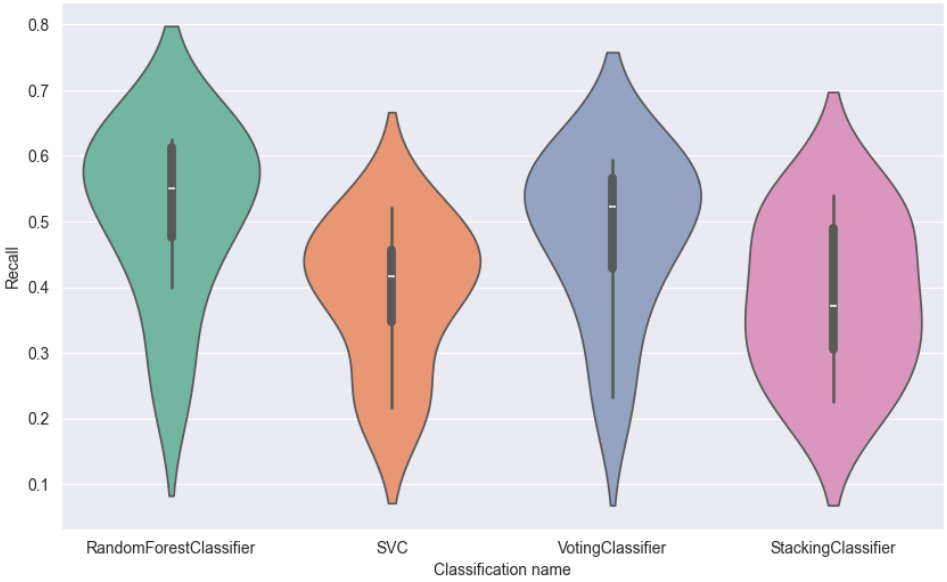


Figure A.5: Violin plot showing the recall distribution across classification models

Appendix B

Data Exploration Graphs

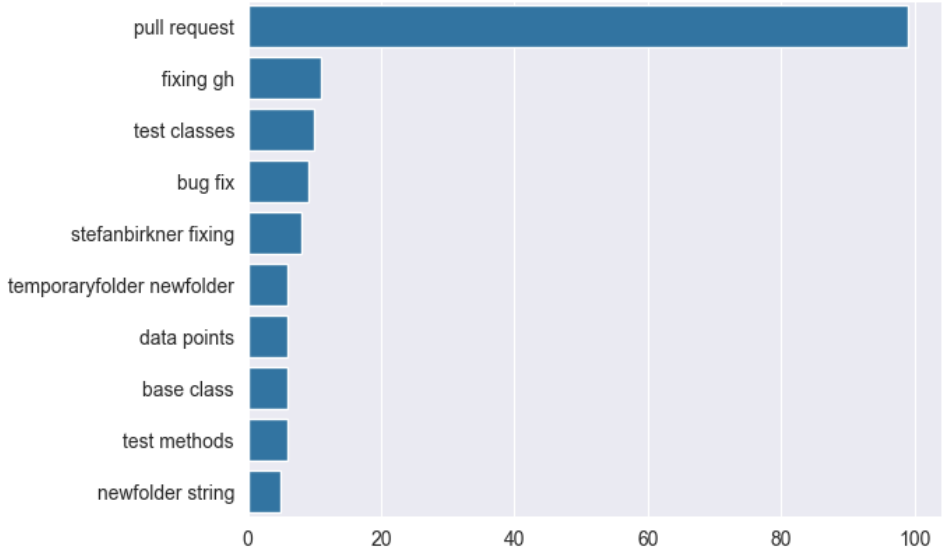


Figure B.1: Most popular bigrams in *Changes* column for JUnit dataset.

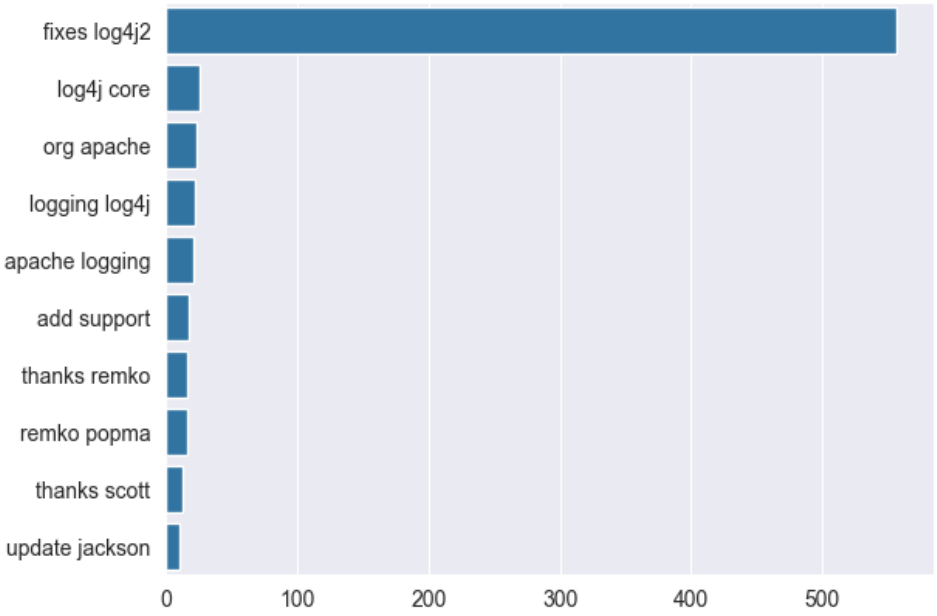


Figure B.2: Most popular bigrams in *Changes* column for Log4j dataset.

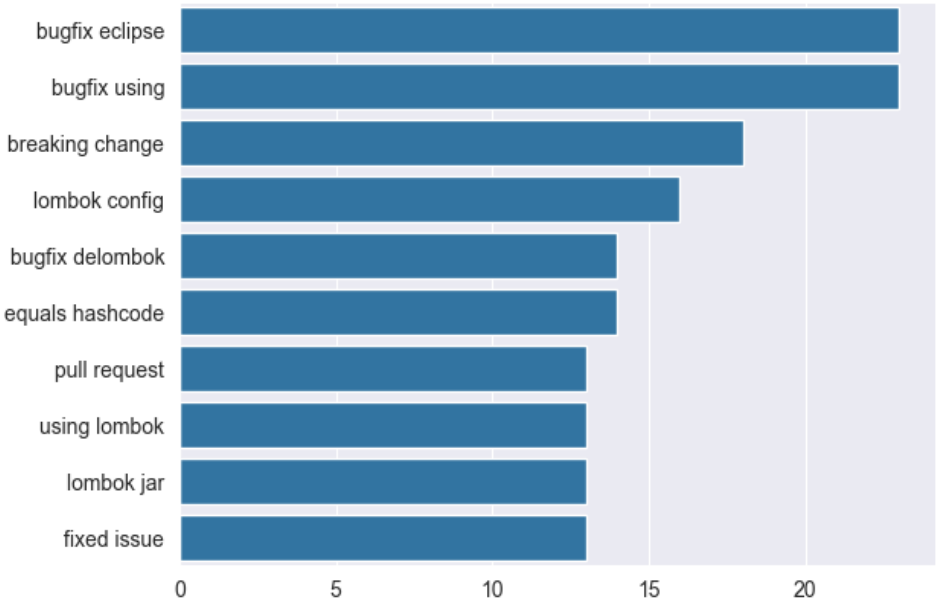


Figure B.3: Most popular bigrams in *Changes* column for Project Lombok dataset.

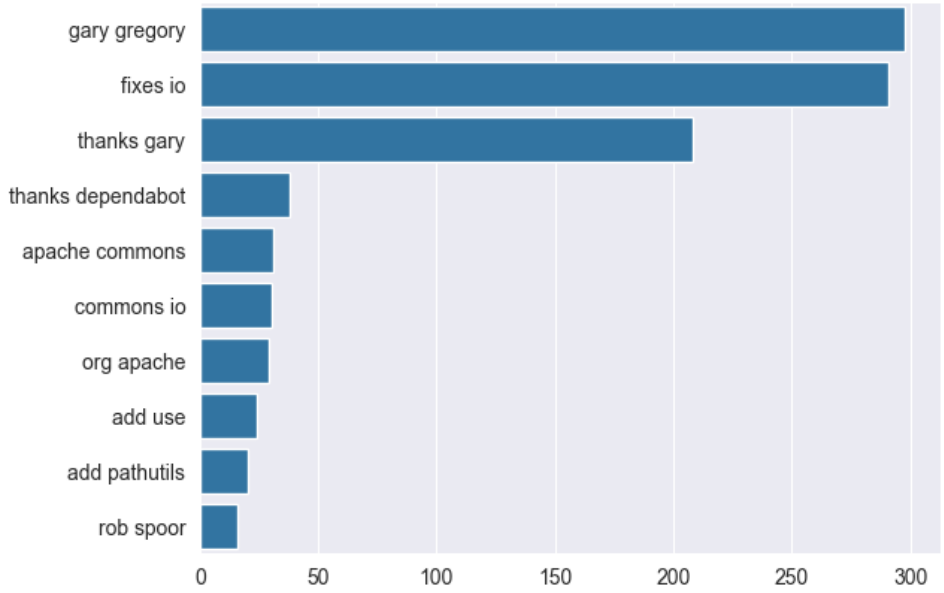


Figure B.4: Most popular bigrams in *Changes* column for Apache Commons IO dataset.

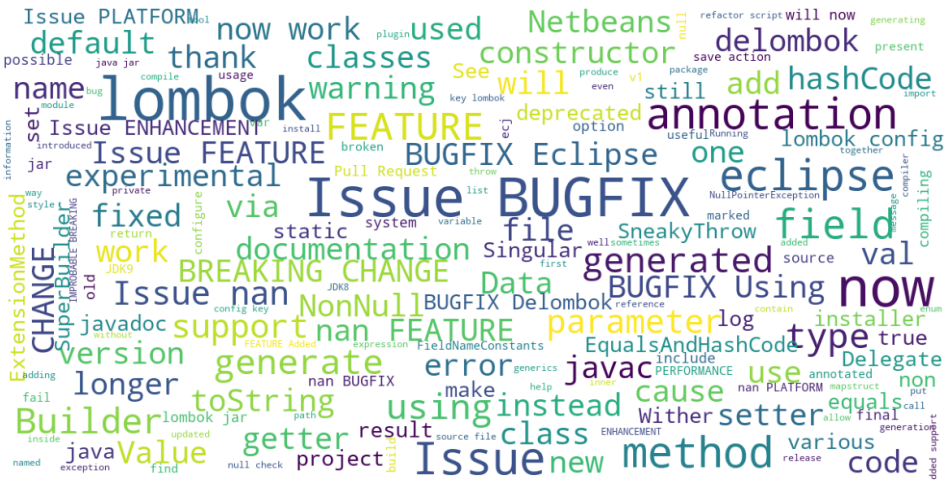


Figure B.5: Word cloud for Project Lombok dataset

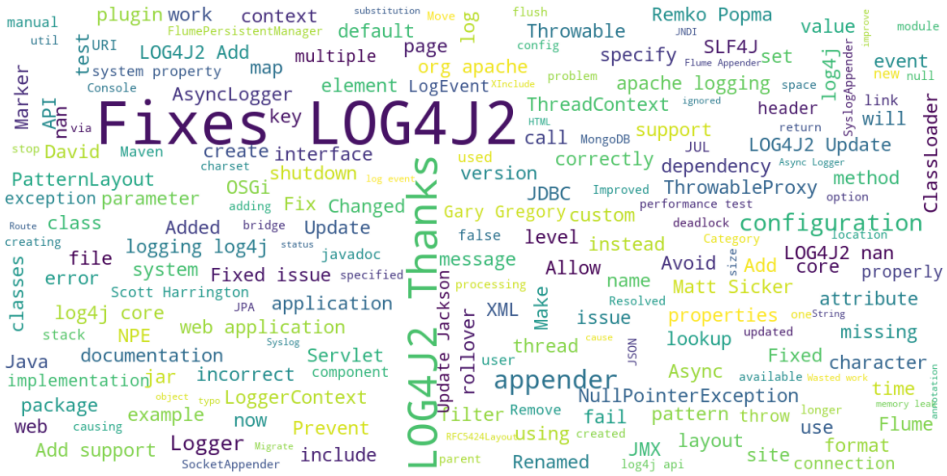


Figure B.6: Word cloud for Log4j dataset

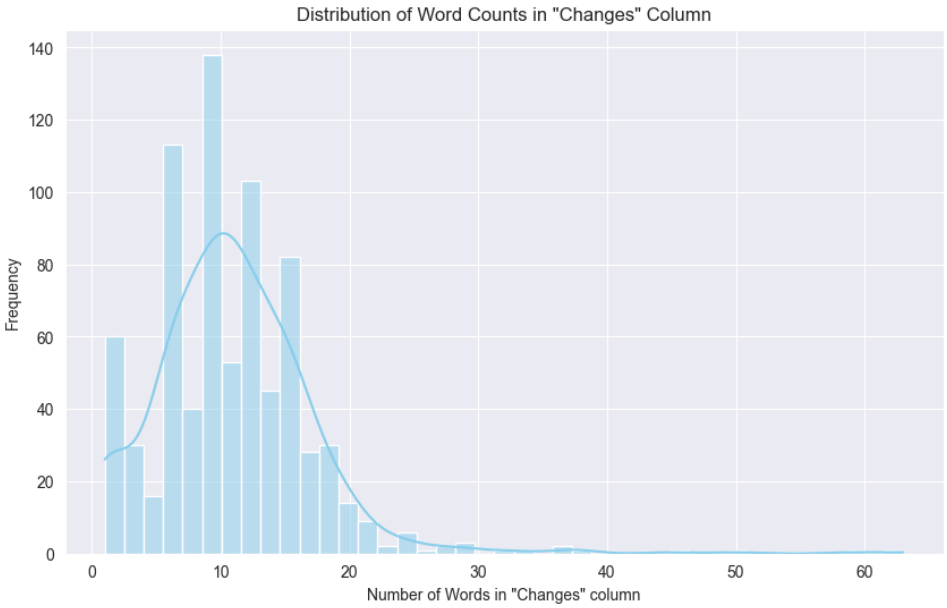


Figure B.7: Text count distribution in Changes column for Apache Commons IO dataset

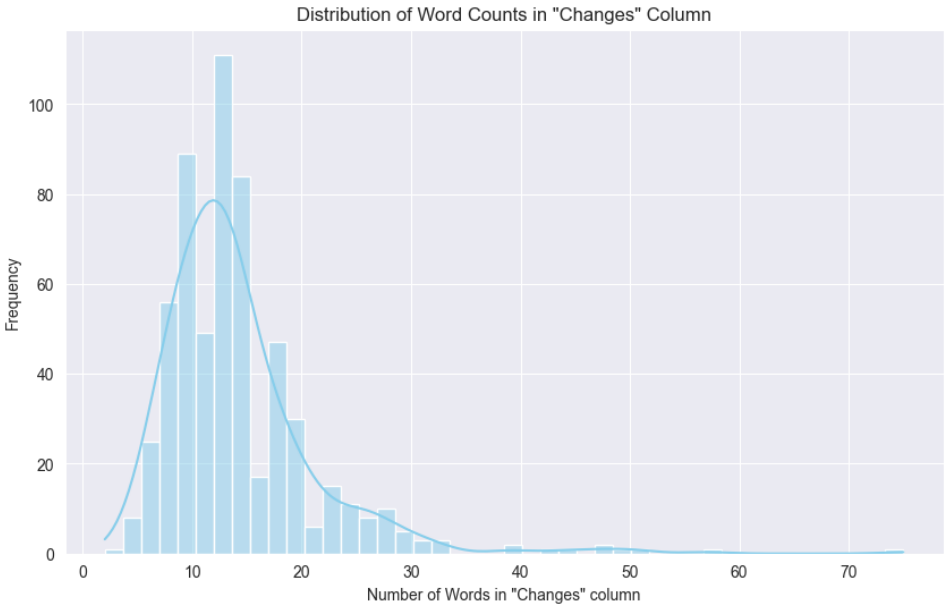


Figure B.8: Text count distribution in Changes column for Log4j dataset

Appendix C

Performance Metrics

Table C.1: Performance of ML models on Apache Commons IO dataset

Classifier	Accuracy	Precision	Recall	F1-score
RFC	0.761	0.646	0.613	0.606
SVC	0.704	0.627	0.435	0.474
VC	0.761	0.648	0.591	0.593
SC	0.725	0.732	0.540	0.579

Table C.2: Performance of ML models on JUnit dataset

Classifier	Accuracy	Precision	Recall	F1-score
RFC	0.562	0.440	0.400	0.400
SVC	0.479	0.304	0.217	0.208
VC	0.542	0.450	0.372	0.369
SC	0.500	0.238	0.226	0.211

Table C.3: Performance of ML models on Log4J dataset

Classifier	Accuracy	Precision	Recall	F1-score
RFC	0.695	0.595	0.615	0.579
SVC	0.661	0.712	0.484	0.528
VC	0.703	0.587	0.558	0.558
SC	0.653	0.704	0.483	0.522

Table C.4: Performance of ML models on Project Lombok dataset

Classifier	Accuracy	Precision	Recall	F1-score
RFC	0.726	0.257	0.254	0.254
SVC	0.690	0.326	0.236	0.228
VC	0.702	0.257	0.232	0.229
SC	0.690	0.323	0.243	0.229