PROGRAM COMPREHENSION CHALLENGES DETECTION FOR PULL REQUESTS WITH MACHINE LEARNING

EDDIE CHIH-JUNG CHIANG

Department of Electrical, Computer, and Software Engineering
The University of Auckland

A thesis submitted in fulfilment of the requirements for the degree of
Master of Engineering in Software Engineering
The University of Auckland
January 2021

ABSTRACT

The importance of program comprehension in software development is such that software engineers spend more time, 35% to 70% of their time, understanding code than writing it. With pull requests (PRs) becoming the default development collaboration model, code review is becoming more common. There is a paucity of studies focusing on the challenges concerning program comprehension. This study aims to provide an insight into those challenges during code reviews, and to, specifically, detect comprehension challenges with the application of machine learning (ML) to analyze a large number of GitHub PR review comments more efficiently, and investigate the common causes of their occurrences to serve as suggestions as to the perils to avoid during code reviews. In this context, comprehension challenges are focused on instances when a code reviewer seeks information on the program behavior, the intention of a code change, who has experience with the code, or why the code is implemented a certain way.

To evaluate how accurately ML can be used to detect comprehension challenges noted in review comments, an ML classifier was developed with the combination of a linear support vector machine with stochastic gradient descent learning and natural language processing of dialogue act classification, stop words removal, lemmatization, and term frequency-inverse document frequency. To investigate the causes of comprehension challenges, a sample of 748 review comments were analyzed through statistical hypothesis tests, and a content analysis was conducted to compare 384 review comments without comprehension challenges and 384 samples with comprehension challenges. The performance results of the ML classifier showed a 74.3% precision and a 66.7% recall. The quantitative and qualitative analysis showed that the lines-of-code-changed metric made no significant difference to program comprehensibility, and review comments with comprehension challenges were associated more frequently with discussions on bottom-up knowledge.

These results suggest that ML can help to detect comprehension challenges and issues and provide analytics to software development teams to support them to prioritize areas of improvement. Furthermore, missing top-down knowledge regarding programming plans is the fundamental reason for many comprehension challenges.

ACKNOWLEDGMENTS

Throughout the course of the research, I have received a tremendous amount of support and constant encouragement.

First and foremost, my supervisor, Dr. Kelly Blincoe, deserves my sincere appreciation for the opportunity to do this work. With her comprehensive knowledge and expertise, she provided me with invaluable guidance in the formulating of the research topic and methodology to conduct the analysis. I am incredibly grateful for her patience, enthusiasm, and mentoring.

Special thanks to Peter Devine, James Tizard, and Sunny Wang for their assistance in conducting the independent manual data coding. Moreover, the entire HASEL group has provided me with constructive feedback and inspiration for my research design.

I also owe an important debt to Fraedom and Visa Inc. for the Education Assistance Program that allows me to study part-time. Our architecture practice lead, Richard Drew, helped me to explore the possibilities and inspired me to improve the work of the software community. My colleagues, Peter Chen, Yi Chen, Zoran Ljubisavljevic, Theodore Teow, Hong Zhang, and Dimitar Zhivkov, engaged in countless interesting software development, data science, and machine learning discussions. Likewise, I want to thank Chao Li and Kevin Li for contributing their personal GitHub access tokens to use for data collection from GitHub for this study without being disrupted by the API rate limit and for their kindness and support that made it possible to complete this study.

Finally, I would like to express my gratitude to my wife, Sally, my boys, Bon and Pontus, and my parents for their help, love, and support; for providing diversion (sometimes much needed) from the academic study; and for sharing the path on my journey. You are the best supporters, friends, and family one could hope for. You mean the world to me.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

GLOSSARY

AI artificial intelligence

API application programming interface

CCM cyclomatic complexity metric

CVCS centralized version control system

DRY Don't Repeat Yourself

DVCS distributed version control system

GDPR general data protection regulation

HCM Halstead complexity measures

IDE integrated development environment

LOC lines of code

ML machine learning

NLP natural language processing

NLTK Natural Language Toolkit

NPS Naval Postgraduate School

PR pull request

REST	representational state transfer
RQ	research question
SGD	stochastic gradient descent
SSL	secure sockets layer
SVM	support vector machine
tf-idf	term frequency-inverse document frequency
UI	user interface

1 INTRODUCTION

Program comprehension is fundamental to a software engineer's job function and plays a pivotal role in the software development process [1]. A software engineer has to have a sufficient cognitive understanding of the source code before building, maintaining, evolving, reusing, or modeling a software system [2, 3, 4, 5].

Currently, software engineers are spending a large portion of their valuable time on program comprehension and re-comprehension. As a result, they have less capacity to focus on innovations, building features, or enhancing software systems [6, 7, 8]. Despite steady improvement in the software development environment, process, and methodologies, program comprehension tools are rarely utilized, or even known, and comprehension strategies are overly dependent on the context [9].

This chapter starts by defining program comprehension and how it relates to the context of software development, then introduces the importance of program comprehension to the software engineering discipline, and also presents the increasing trend of pull-based development and the role that program comprehension plays in pull request (PR) code reviews. Thereafter, previous related studies were briefly reviewed, and a knowledge gap was identified. Finally, the purpose of this new research is established, the research question (RQ) listed, and the contributions of this study clarified.

1.1 DEFINITION OF PROGRAM COMPREHENSION

Historically, the term "program comprehension" describes the ability to characterize the structure of a software program, its input(s) and output(s), effect(s) during and after execution, and the coding syntax written in the source code of the program [10].

To illustrate the definition, a simple example of a code snippet follows:

```
public double MyCalculation(double a, double b)
{
  c = a + b;
  return Math.pow(c, b);
}
```

A software engineer should be able to understand the intent, sequence flow, and effects of the code snippet and be able to express it in human-level communication:

1. A method that takes two inputs, `a` and `b`.
2. Performs addition to those two inputs and assigns to a local variable `c`.
3. Raises `c` to the power of `b`, using a built-in class.
4. Returns the result to the caller of the method.

Moreover, program comprehension is not only required for source code written by a human but also necessary for computer-generated artifacts, such as templates generated from a code generator or codebase structure created by executing a scaffolding command. The relationship between program comprehension and the different parts of the software development process is illustrated in Figure 1.1.

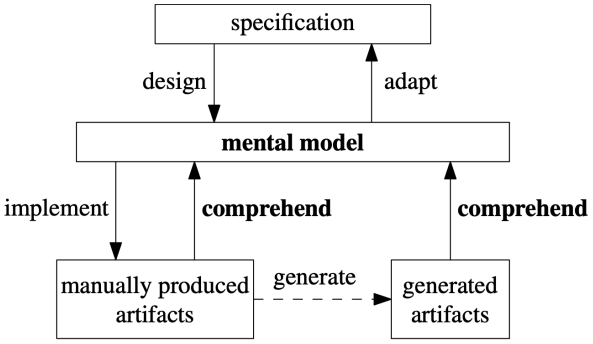


Figure 1.1: Software development process and program comprehension (Reproduced from [11]). Solid lines depict human activities; dotted lines depict automated activities. Bold text depicts the research area of program comprehension.

1.2 IMPORTANCE OF PROGRAM COMPREHENSION

One aspect of the importance of this topic is the cost in time to software engineers. Evidence suggests that software engineers may spend more than 50% of software maintenance capacity in reading and understanding source code [8, 12, 13]. Similarly, one of the approaches software engineers use for program comprehension is to navigate through call stacks or relevant code segments. A study that tracked the screen activity found that, on average, 35% of their time was spent on this activity [7]. Another

study collected interaction data from an integrated development environment (IDE) and revealed that approximately 70% of time spent on coding or revising code could be categorized in program comprehension [6]. In addition, they are likely to spend 3.5 times more time on understanding the source code than reading the supporting documentation and spending only a moderate amount of time on the actual implementation of the enhancement [8]. Albeit that the findings of different studies regarding the percentage of time spent on program comprehension vary between 35 and 70% [6, 7, 12, 14, 15, 16], it is well established from a variety of research that software engineers spend more time on understanding code than on writing it [6, 7, 8].

The second aspect is the cost of resources to organizations. For a successful software system, change is intrinsic and inevitable [5]. Unless an organization actively attempts to maintain or reduce a software system's complexity, it will increase in complexity as it evolves [17]. The human mental capacity to process large and complex codebase is limited. Code-level complexity can affect the readability and understandability of the codebase, resulting in more time and effort spent to accomplish a development task. For example, too many nesting levels or cyclic dependencies in a method will affect the readability, rendering it difficult to understand [18, 19]. In addition, software maintenance often accounts for a large proportion of an organization's budget [20], even up to 50 to 90% [21, 22]. Therefore, program comprehension is estimated to cost an organization between 30 and 50% of the operating expense [6].

The third aspect is the cognitive load of software engineers. This load increases as the software system complexity increases [23, 24]. Humans have limited cognitive capacity and can only memorize a certain amount of information during any given time frame, and familiarity decreases as time passes. This "forgetting factor" results in additional effort to re-familiarize even previously seen source code [18, 24, 25] and leads to a phenomenon in which software engineers sometimes attempt to copy and paste source code to avoid too much effort, both in time and mental exertion [25, 26, 27]. For example, suppose a software engineer cannot understand all the possible side effects of a change; instead of refactoring a code block into a reusable method, the propensity is to copy and paste. Often they postpone refactoring the code until they had copied the same code multiple times [9, 27]. Consequently, the cloned source code violates the "Don't Repeat Yourself" (DRY) principle, rendering the software system more difficult to maintain. For any fix or enhancement required on the code

snippet, multiple copies are scattered throughout the codebase, increasing the risk factor and reducing the system integrity and consistency [26, 28, 29].

1.3 MOTIVATION AND OBJECTIVES

The first in-depth analyses of program comprehension emerged during the late 1970s [13, 30, 31]. There have been many improvements in the software development environment over the ensuing 40 plus years. For example, some IDE now provides artificial intelligence (AI)-assisted development, uses context from the codebase, and combines with machine learning (ML) for more intelligent code completion.[1] These changes have streamlined software development tasks for software engineers. However, it still takes a notably large portion of time to understand code [32], and program comprehension tools are only minimally used [9].

To date, few studies have investigated how and why these improvements assist software engineers [9, 11, 32]. Robillard *et al.* [33] conducted an exploratory study in a laboratory setting with five developers.[2] They concluded that it is more productive to investigate a codebase with a systematic strategy than with an unmethodical approach. Maalej *et al.* [9] studied 28 developers across seven companies. They found that the comprehension strategy used depended significantly on the current work contexts, such as task at hand, knowledge on the codebase, development experience, types of technology, or framework used.

Although extensive research has been conducted on the actions of developers in development tasks to understand the codebase as part of the tasks, few have focused on the specific program comprehension challenges developers face or why they occur.

1.3.1 *Definition of Program Comprehension Challenge*

In this study, comprehension challenge is defined as when a developer seeks information on the program behavior, the intention of a code change, who has experience with the code, why the code is implemented a certain way, or who wrote this piece of code [9, 14].

1 Visual Studio IntelliCode, https://visualstudio.microsoft.com/services/intellicode.
2 Throughout the context of this study, the terms "software engineer" and "developer" are interchangeable.

1.3.2 *Program Comprehension and Pull Requests*

Pull-based development is a software development collaboration model. Code review processes and PRs are at the center of this model. A software engineer can initiate code changes by creating a PR for someone to review the changes before merging them into the branch. Those code review comments are conveyed and captured as written communication associated with the PR and code position. With more than 56 million developers as of 2020, GitHub is one of the largest developer platforms supporting pull-based development [34].

Moreover, pull-based development has been trending during the last ten years [35, 36] and is expected to become the default software development collaboration model [37]. Gousios *et al.* [36] have classified the frequency with which developers created PRs and surveyed the main themes of challenges that emerged from the PR process. The summarized findings of the specific challenges developers face with the pull-based development are presented in Figure 1.2. The thickness of a line represents the number of responses. Unsurprisingly, understanding codebase is one of the top three most frequent challenges faced by developers.

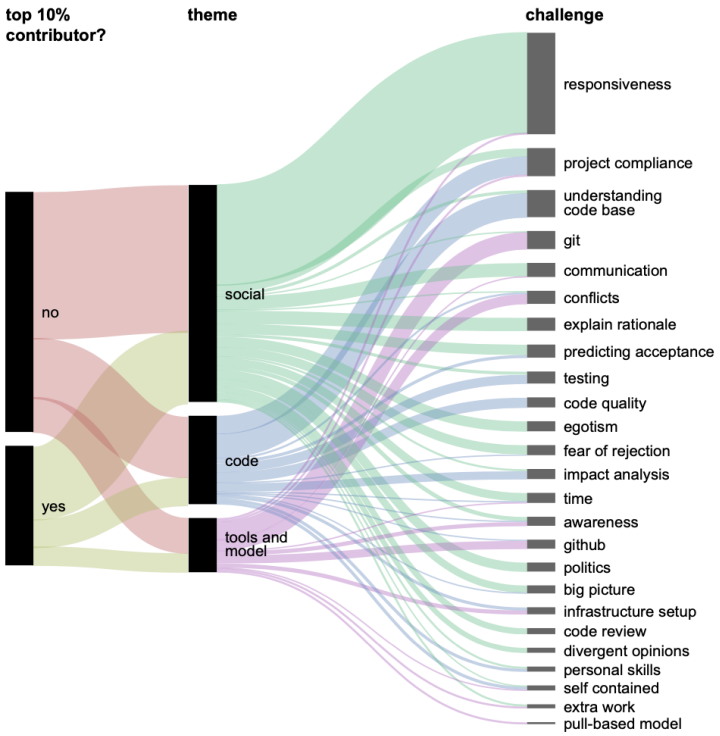


Figure 1.2: Challenges with the pull-based development (Reproduced from [36]).

1.3.3 *Research Scope*

As the pull-based development model is progressively becoming more popular, more software engineers are performing code reviews. In addition, Gousios *et al.* [36] observed that, close to 70% of the time, software engineers would communicate intended code changes by opening a PR, and that program comprehension is one of the most frequent challenges that code reviewers face. A search of the literature revealed a paucity of studies that investigated written software engineer communications in PR code reviews.

A gap in the literature that motivates the need and establishes the importance of studying program comprehension during PR code reviews has been revealed. This study aimed to identify the accuracy of detecting program comprehension challenges faced by software engineers during code reviews with the application of ML, analyze the specific challenges surrounding program comprehension to compare to the challenges of software maintenance, and discover the common causes of the challenges. Therefore, this study investigates the following four main RQs:

RQ1. What are the attributes of pull requests that affect program comprehensibility?

RQ2. How accurately can a machine learning classifier detect program comprehension challenges in a pull request code review?

RQ3. What types of program comprehension challenges do reviewers face?

RQ4. Why do reviewers face program comprehension challenges?

This study systematically analyzed data of PR review comments, aiming to identify instances when software engineers encounter program comprehension challenges and the context of those challenges. Both qualitative and quantitative methods were used in this study, with PR review comments data gathered from two main sources: GitHub REST API v3[3] and GHTorrent[4].

Furthermore, this study explicitly investigated instances where developers encountered challenges in understanding the codebase and also advanced towards improving program comprehension support for software engineers. A substantial amount of code review comment textual data were collected, and ML and natural language

3 GitHub REST API, https://docs.github.com/v3.
4 The GHTorrent project, https://ghtorrent.org.

processing (NLP) were leveraged to analyze those comments. Many existing studies with adequate feature engineering have proven that ML is recognized as a reliable approach for data analysis. This study developed an ML classifier to classify the PR review comments, and to detect when a code review comment expresses difficulties in understanding code changes. In conjunction with manual content analysis, this study further identified common types of challenges and discussion topics during code reviews.

1.4 CONTRIBUTIONS

This study aims to provide a better understanding of the specific challenges to program comprehension and gain a deeper understanding of software engineers' particular needs to understand the codebase when facing a challenge. The novel contributions of this study are:

1. The quantifiable attributes of a PR were analyzed, and their relationship with the program comprehensibility of the code changes in the PR and the possibility that some of the well-known software complexity metrics apply to PRs are discussed.

2. Various ML algorithms, training approaches, and NLP techniques are discussed; their performances in detecting comprehension challenges during a PR code review are compared; and a method of using an ML classifier for promptly detecting program comprehension challenges in PR code reviews with acceptable performance is provided.

3. A data aggregator developed in Python[5] for collecting PR data from GitHub and GHTorrent is publicly shared.

4. All developed Python ML scripts[6] are provided and the raw and processed PR comments data[7] shared as a replication package for future studies.

5. A comparison of the types of program comprehension challenges during code reviews and software maintenance is discussed.

5 https://www.python.org.
6 Code Comprehension Classifier for Pull Request Comments (CCC4PRC), https://github.com/eddie-chiang/ccc4prc.
7 Datasets 20190503, https://bit.ly/2JvMhC1.

6. The common causes of program comprehension challenges during code reviews are presented.

The significance of these contributions enables more efficient code review and pull-based development. The ML classifier provides an in-depth understanding of whether AI-assisted tools can be used to assist software engineers with reviewing code changes, which, consequently, allows the more efficient onboarding of new team members, minimizes the time spent on trivial impediments affecting program comprehensibility, and maximizes output with more development velocity. Finally, knowledge of the common types of information needs can enable suggestions for avoiding some perils when software engineers creating PRs, turning the code review into a more enjoyable and productive exercise. Decreasing the reviewers' cognitive load will enable them to focus more on ensuring code quality and minimizing bugs, and, therefore, reduce the overall cost of software development.

1.5 STRUCTURE OF THE THESIS

The remainder of this thesis is structured into six chapters:

CHAPTER 2 explains the background and provides the reasons for researching program comprehension with the implementation of PR and ML.

CHAPTER 3 discusses the trends, approaches, and evolution of the field in the literature review. These are then interpreted and evaluated to connect to the research conducted in this study.

CHAPTER 4 describes the detailed design of the research methodology and explains why the method was adopted.

CHAPTER 5 presents the results and findings related to the RQs.

CHAPTER 6 delves into the key findings, their relevance to the literature review and the RQs, and considerations on the validity of the findings of this study.

CHAPTER 7 concludes this thesis with a conclusive discussion of the findings, main contributions, and future research recommendations.

2

BACKGROUND

To frame the discussion in this study, it is crucial to describe the basic components of a PR, clarify how and why code reviews are conducted in a PR, and justify the use of ML for analyzing code review comments.

2.1 PULL REQUEST

A source code file is tracked with a change history on every commit or check-in in version control systems. Git is a distributed version control system (DVCS), and since its inception in 2005, it has revolutionized the way software engineers collaborate on software development [38].

In addition to the more traditional centralized version control systems (CVCSs), such as Team Foundation Version Control[1] or Subversion[2], a DVCS allows more flexible and rapid branching or forking (i.e., cloning) of the codebase. Every branch created in a CVCS requires contacting the central server. The server then has to process the request, allocate data storage, and permission check to ascertain whether the user has the appropriate authorization for the desired destination branch location.

In contrast, a DVCS is decentralized and has no such constraints. Each commit is associated with a global identifier across branches and repository forks. A software engineer can create a new branch locally on their computer or fork a repository in their repository hosting. All of this can be achieved while preserving the commit history and change tracking that can be referenced across branches or repositories. The ability to distribute version control enables the pull-based development model.

Consequently, a software engineer can propose changes as a PR, and it is up to the repository owner or reviewers to accept or reject the PR [35]. A PR can be utilized for feature development, bug fixes, intra-repository collaboration within the same team by creating separate branches, or inter-repository collaboration with the community by creating forks.

1 Team Foundation Version Control, https://docs.microsoft.com/en-us/azure/devops/repos/tfvc/what-is-tfvc.

2 Apache Subversion, https://subversion.apache.org.

A classic example of a PR starts with a software engineer working on a new feature, identified as the code change author. To minimize disruption and risk on the main branch, the author creates a snapshot of the main branch as a feature branch, allowing them to work on it independently. They make changes to the feature branch until the feature is complete and ready to be merged back into the main branch. As a workflow for code review from peers, the author creates a PR. One or more reviewers inspect the change and may make some comments. On GitHub, those comments can be tied to a specific code position, referred to as PR review comments, or be a global comment regarding the entire PR, referred to as PR issue comments. The author then reads those comments and makes amendments to address the feedback. This process can be iterative, with multiple rounds of code review and additional commits. Once the reviewer approves the code review and the PR, the feature branch merges back into the main branch. Figure 2.1 illustrates the interactions among authors, reviewers, the main branch, and a feature branch during a PR process.

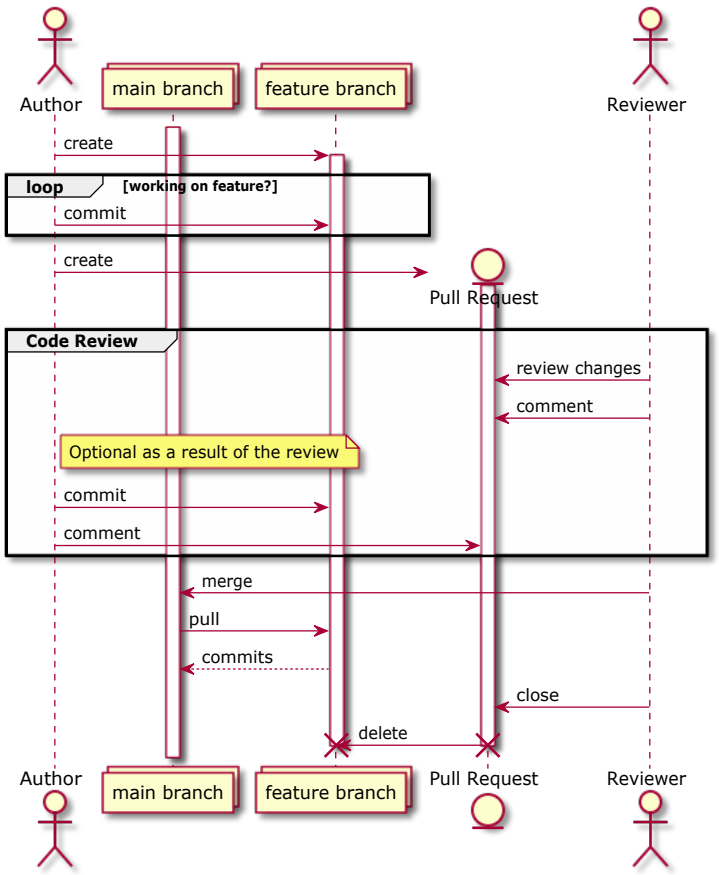


Figure 2.1: A pull request sequence.

2.2 CODE REVIEW

Code reviews typically involve reviewers, excluding the author, inspecting the code changes manually. The ultimate goal is to improve code quality, identify any issues, and provide feedback to the author so that they can amend accordingly before approving the PR [39, 40], while some code reviews are motivated by code ownership and knowledge sharing [40, 41].

Figure 2.2 shows a sample screenshot of a code review discussion thread during a PR process on GitHub. Although details for different code review tools from the different platforms may differ, the core features remain largely similar [42]. The reviewer provides feedback, referencing the changed code and the rationale for the suggested change. The author acknowledges the feedback and updates the code accordingly.

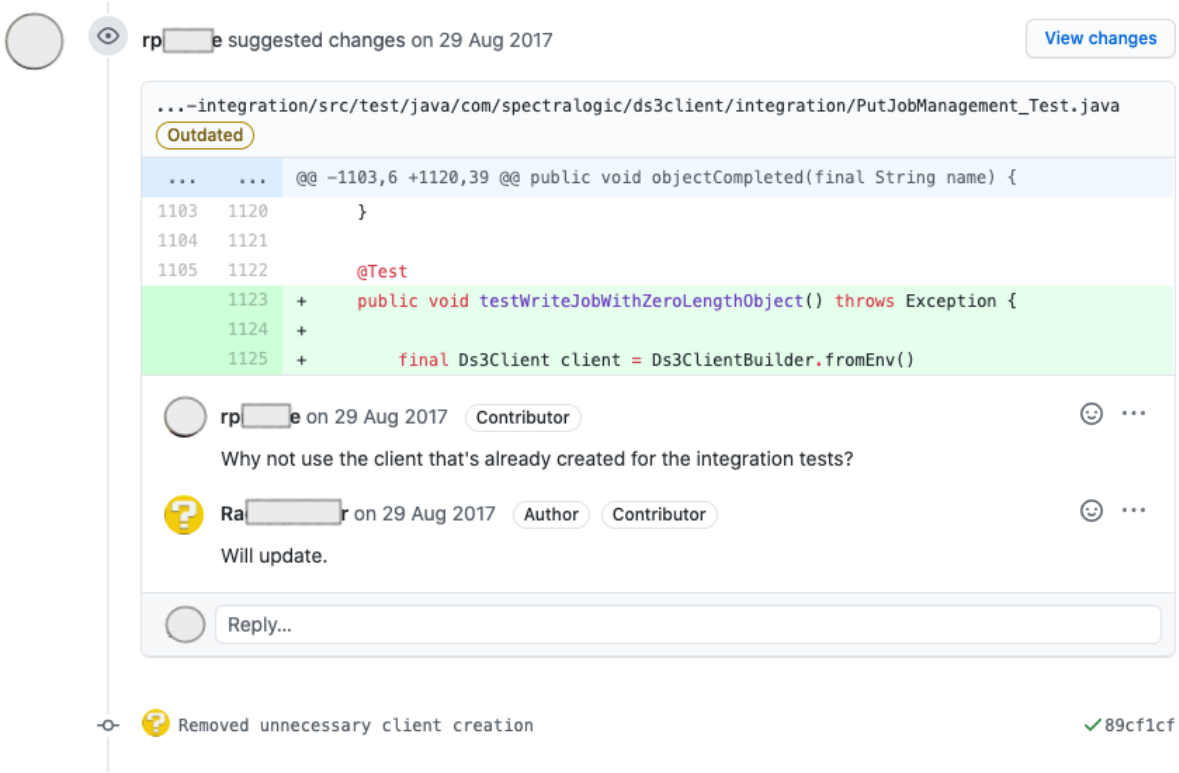


Figure 2.2: Code review discussion thread in a pull request on GitHub.

The following list describes the relationship between the domain objects relevant to a PR captured on GitHub that are within the scope of this study:

- An author or a reviewer are both subtypes of a user.

- An author is a user who creates a PR.
- A PR contains one or more commits.
- Each commit contains changes to one or more files.
- A reviewer can comment overall regarding the entire PR; this is referred to as a PR issue comment.
- A reviewer can comment on a specific code change in a file; this is referred to as a PR review comment.
- An author can respond to a PR review comment by creating a subsequent PR review comment in the discussion thread.

The entity-relationship model is illustrated in Figure 2.3.

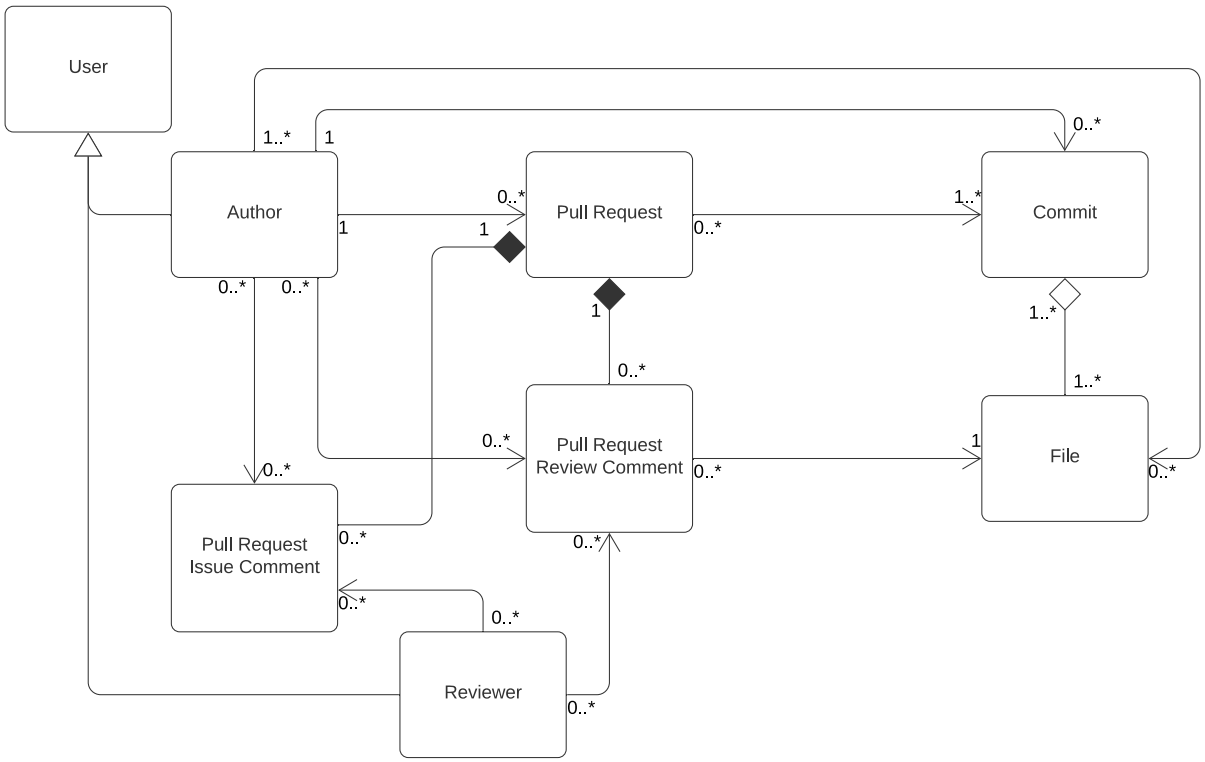


Figure 2.3: Pull request review comments entity relationship modeling.

2.3 MACHINE LEARNING AND NATURAL LANGUAGE PROCESSING

Today, with the pull-based development, code reviews are becoming more tool-based, asynchronous, and communicated in written form [41, 42, 43]. Code reviewers are motivated to ensure code quality, mainly by inspecting and reading through the code changes [39], yet there is a lack in instantaneous interaction and discussion with code authors to guide the code changes. In addition, one of the most significant challenges during code review is program comprehension [36, 41]. This makes code review comments the ideal data source for studying program comprehension.

However, with the large amount of data available from GitHub and given the bandwidth constraint, it is not humanly possible to analyze even a fraction of that data. Fortunately, ML offers methods to process large amounts of data efficiently. An ML model with an appropriate algorithm, a training dataset, and extracted features relevant to the problem domain can build a classifier for predictions. The classifier could assist in identifying specific occurrences when a reviewer fails to understand a code change or its rationale. This means that researchers will have more time to focus on exploring the causes of those program comprehension challenges.

Furthermore, NLP extracts and filters features of code review comment textual data for ML algorithms to evaluate. NLP is a subset of ML, and it concerns the understanding of the natural language used by humans [44]. The following is a simple sentence to illustrate:

> *Today it is hot and sunny.*

One of the most basic tasks to process the sentence is to perform tokenization. This means delimiting words in the sentence into meaningful units with spaces, and the result would be: "*Today*," "*it*," "*is*," "*hot*," "*and*," "*sunny*." There are various methods to achieve the goal in NLP. In real-world use cases, the tasks and processing involved are often much more complicated.

With the maturing development and increasing focus on NLP and ML in recent years [44, 45] and the development of ML models that show promising prediction performance in text analysis in other studies [46, 47], it is being increasingly regarded as a reliable technique to automate and reduce the effort required for analyzing code review comments.

3

LITERATURE REVIEW

Research into program comprehension has a long history, and much of the research relevant to this study has been anchored in three major dimensions:

1. Studying the applied program comprehension models and strategies.
2. Inventing different metrics and techniques to measure program comprehension.
3. Identifying information needs and types of challenges faced.

This chapter reviews each of these dimensions, including outlining key concepts and theories, exploring applications of those concepts and theories, and highlighting and comparing the approaches and techniques applied in similar studies. The purpose of these reviews is to examine whether the RQs are pointing in the right direction to advance one more step in this research field.

3.1 PROGRAM COMPREHENSION MODELS

Different theories regarding program comprehension models exist in the literature. These models portray the mental activities involved and how software engineers memorize information to understand code, and provide insight into what information they search for in their minds to construct the functions of the program [48, 49].

There are more than seven relatively complex models described in the literature [49, 50], but most are comprised of two basic processes: top-down or bottom-up [32]. This section discusses those two fundamental processes and an integrated model that combines both of those processes.

3.1.1 *Top-down Comprehension Model*

A top-down comprehension model starts when a software engineer has a high-level understanding of the program's context. The software engineer assumes a behavior or feature, either functional or non-functional, and then analyzes the codebase to find

the relevant patterns, code blocks, or beacons to either support, reject, or further refine the assumption. In this process, they focus only on the relevant beacons to validating the assumption; any irrelevant information is discarded [49, 51].

In a simple example, a software engineer familiar with the mobile push notification system first makes an assumption on how a push notification is delivered, in the following sequence:

1. The back-end system generates a notification and sends a web service call to a third-party notification service.

2. The notification service delivers the push notification to the end-user's device.

With this assumption in mind, the engineer delves into the code, first by finding the implementation where the notification is generated and then following the sequence flow to search for web service calls. Any unrelated information, such as a stored procedure to audit the notification in a data store, does not verify the assumption; thus, it is ignored. Once the web service call to the notification service is found, the assumption is validated and the goal is achieved by using the top-down model.

The comprehension model is more intuitive than other models to software engineers. When they are familiar with the application domain, the approach is far more efficient than reading the code line by line. They can recognize relevant beacons more easily due to the knowledge they possess on the code structure or the architecture of the software system [51].

3.1.2 *Bottom-up Comprehension Model*

In contrast to the top-down model, a bottom-up comprehension model is applied when a software engineer does not have sufficient understanding of the application domain. Without that knowledge, they cannot recognize the relevant patterns, code blocks, or beacons. In this situation, they carefully read through the code to understand the implementation details and glue together semantically relevant pieces to form a more significant chunk of information, which is a higher-level abstraction. Thereafter, they repeatedly aggregate more chunks until they can articulate a sufficient level of understanding to address the task at hand [48, 52].

To revert to the previous example of the mobile push notification system, a software engineer may receive an error alert with an error message (e.g., an HTTPS handshake

failure) from an application monitoring system and a call stack. The task is to determine the source of error, its cause, and any impact on the end-users. The software engineer can attempt to comprehend the system in the following way:

1. Examine the error message.
2. Locate the most recent method call.
3. Proceed to the next method in the call stack.
4. Repeat until the references to the method can be narrowed down and match any preceding log entries.

These diagnostic steps aid in the formation of chunks of information, and each chunk can help the software engineer to discover more about what the system does, why the error was occurring, and the consequences and impact radius caused by the error. After repeating several iterations of these steps, they may be able to determine that the system failed when sending a push notification to a mobile device because of an expired SSL certificate, used for web service integration with the third-party notification service, and the repercussion is that the end-user may not have received the push notification. The above example demonstrates the bottom-up approach to gain a sufficient level of understanding in action.

3.1.3 *Integrated Comprehension Model*

It is almost impossible for anyone to know everything at all times with large and sophisticated software systems. Hence, when the program is familiar, the comprehension process often involves both top-down and bottom-up approaches [53]. A software engineer may have knowledge of the application domain, but, when there is unfamiliar code to be understood, they may have to combine the two approaches.

The integrated comprehension model by von Mayrhauser and Vans [53] describes such mental activities. It refers to the program model as the bottom-up approach to reading through code to gain programming domain knowledge. It also refers to the situation model as the subsequent step to develop in addition to the program model to form problem domain knowledge regarding the functional feature. In short, the bottom-up and situation models combine with the top-down model to establish an understanding of the unseen code, as illustrated in Figure 3.1.

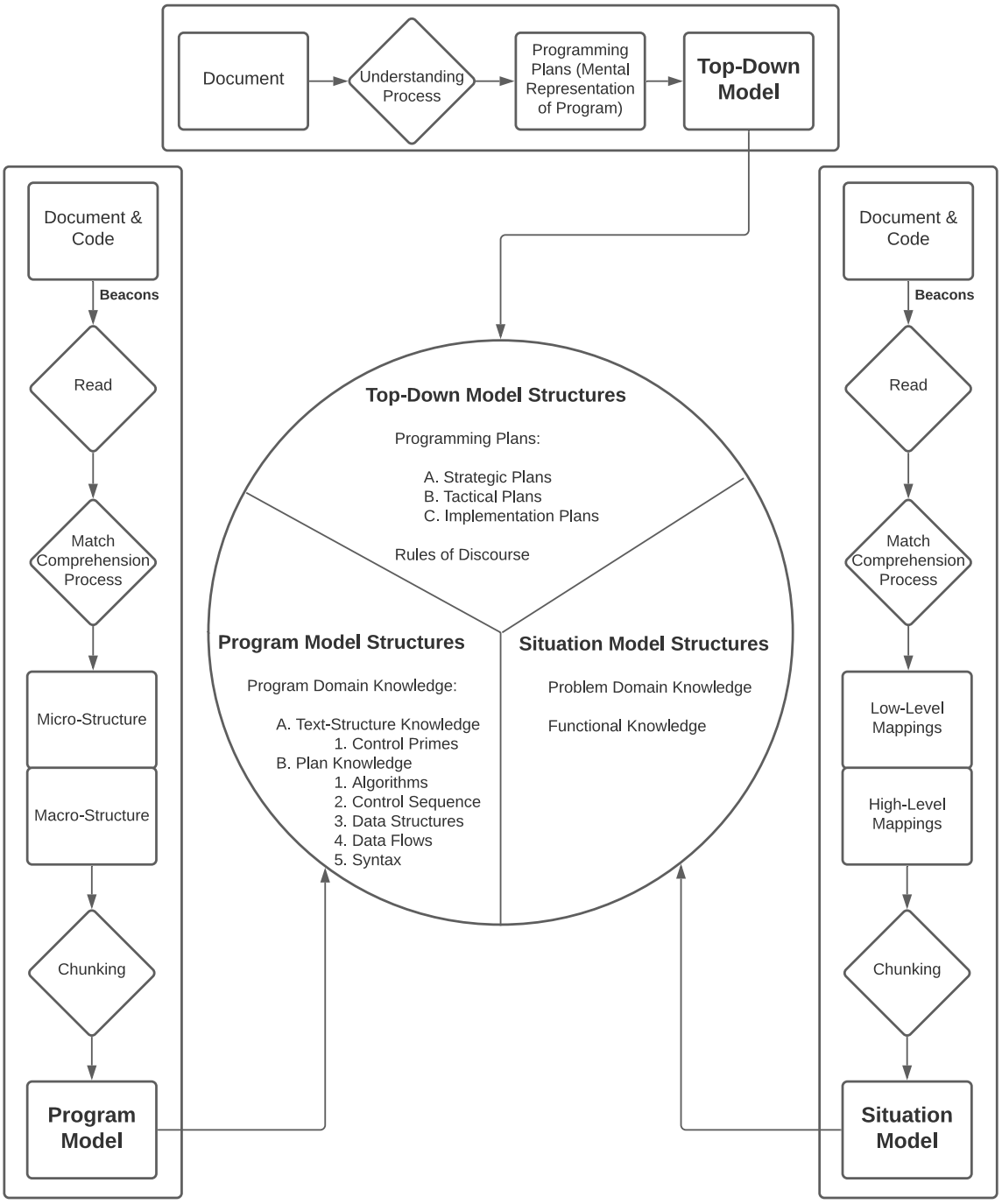


Figure 3.1: Integrated comprehension model (Redrawn from [53]). This depicts how top-down, bottom-up, and situation models combine to form the required knowledge.

To once again revert to the previous examples on the mobile push notification system, some new feature that generates actionable alerts has been developed and it

is unfamiliar to the software engineer. The integrated comprehension model could be implemented as follows to understand the new code:

1. Read through the newly implemented method.
2. Gain an understanding of the data structure for the notification payload.
3. Be aware that the system did not previously support actionable alerts.

As a result, they are able to map the newly implemented code for the actionable alerts feature with the help of the existing application domain knowledge.

RELEVANCE TO THIS STUDY These studies have collectively outlined critical observations regarding the intellectual processes during program comprehension, providing insight into what information software engineers seek.

Although there is a tendency among software engineers to prefer the top-down approach, as it requires less cognitive input, it is well established that different problem contexts may lead to various activities to solve the problem [51]. In addition, the characteristics of individual software engineers, such as ability, knowledge, and experience, are the drivers of the differences in dealing with comprehension tasks [48, 54, 55], and they frequently switch between different models, depending on the level of known information or details required [53]. Thus, rather than contradicting each other, these models complement each other to fill any gaps.

This study concentrated on those three well-established comprehension models. Consideration of the comprehension model(s) a reviewer implements during a PR code review process can aid understanding of the types of information a reviewer needs. Conversely, analyzing the reviewers' information needs to find emerging themes can aid understanding of the potential comprehension model(s).

3.2 MEASURING PROGRAM COMPREHENSION

There are two aspects to measuring program comprehension. One aspect is to develop quantifiable code metrics to represent software complexity, such as lines of code (LOC), or a cyclomatic complexity metric (CCM). Another aspect is to measure how well human subjects understand source code.

In the past, most studies explored quantifiable code metrics based on software systems' characteristics to measure the complexity. Code-level complexity has a

correlation to the program's comprehensibility. The more complex the codebase, the more difficult it is to understand [18, 19]. Moreover, a widely accepted metric of software complexity can assist in the evaluation of the quality of software and provide more confidence in the estimation of software development [56].

However, the generalizability of the published metrics is problematic and they may not reflect software complexity correctly. Thus, other researchers have conducted various empirical evaluation approaches to measure cognitive processes, predominantly comprehension tasks, memorization, and think-aloud protocols [31, 51, 52, 53, 57, 58, 59].

The two aspects, aiming to measure the comprehensibility of the software system and investigating different attributes, properties, or characteristics that could affect software engineers' understanding of the code, are two sides of the same coin. This section discusses and evaluates the more common code metrics and the trends and approaches to measuring cognitive processes.

3.2.1 *Code Metrics*

There is a considerable amount of literature on different code metrics that have been invented to quantify a software system's attributes, properties, or characteristics in a numerical rating. Most of these metrics lack a substantial theoretical foundation and are, instead, based on practical experience. The academic literature on these metrics revealed the emergence of several contrasting themes regarding the correlation between those metrics and program comprehensibility.

3.2.1.1 *Lines of Code*

The LOC is one of the most easily calculated measurements. As the name suggests, it counts the number of lines in the source code, including comments or empty lines. A study by Herraiz and Hassan [60] suggests that LOC is a useful metric to estimate the effort required to understand the codebase. This metric is also easy to understand and can be more easily communicated to non-technical stakeholders. On the contrary, Ajami *et al.* [61] found a different perspective in their study, which in some cases, a longer code snippet can take less time to understand than a shorter one, and they advised LOC may not be the best metric to measure comprehension. Furthermore,

LOC does not consider the complexity of each line [62], as in the example shown in Listing 3.1.

Listing 3.1: An example of lines of code in which comprehensibility is not weighted equally.

```
int i = 1;
i = (++i) + (++i);
```

The second line is considerably more complex than the first line and requires significantly more cognitive load to comprehend. In LOC, comprehensibility does not always weigh equally.

3.2.1.2 *Halstead Complexity Measures*

Another popular metric is the Halstead complexity measures (HCM) proposed by Halstead [63] in 1977. It measures the number of operators and operands in the code and offers several metrics, with difficulty being one of those metrics related to the difficulty of understanding code. The difficulty metric, D, is calculated by the following equation:

$$D = \frac{n_1}{2} \times \frac{N_2}{n_2},$$

where n_1 is the number of unique operators, N_2 is the total number of operands, and n_2 is the number of unique operands.

In the first line of Listing 3.1, there are three unique operators, namely `int`, `=`, and `;`. The total number of operands and the number of unique operands are both two: `i` and `1`, so then,

$$\begin{aligned} D_{\textit{line one}} &= \frac{3}{2} \times \frac{2}{2} \\ &= 1.5. \end{aligned}$$

As for the second line, the number of unique operators is five: `=`, `()`, `++`, `+`, and `;`, the number of unique operands is one: `i`, and the total number of operands is three, then,

$$\begin{aligned} D_{\textit{line two}} &= \frac{5}{2} \times \frac{3}{1} \\ &= 7.5. \end{aligned}$$

The difficulty metric for the first line is 1.5, and it is 7.5 for the second line.

Compared to LOC, where each line weighs equally, HCM offers a superior grasp of the relative complexity in this example. It is a more sophisticated measurement yet only requires static code analysis and is still relatively easy to measure. However, HCM lacks execution analysis and neglects the software system's control flows and structural properties, such as method calls jumping to a different class file, and cannot always be considered a sufficient measure [62, 64].

3.2.1.3 *Cyclomatic Complexity Metric*

In contrast to HCM, CCM analyzes the structural properties and the control flows. The CCM is one of the few of all the code metrics that has a theoretical basis. McCabe [65] invented the metric back in 1976 based on graph-theoretic complexity, which analyzes the decision structure of a software system and measures the possible paths and combination of execution. The cyclomatic complexity, M, is defined as

$$M = E - N + 2 \times P,$$

where E is the number of edges, N is the number of nodes, and P is the number of connected components.

For example, the following code can be associated with a program control flow activity diagram, as illustrated in Figure 3.2.

```
if (thesis finished?) equals (yes) then
:celebrate;
else
:keep calm and carry on;
```

There are four edges and four nodes. Since it is a simple code snippet and not calling for another method, there is only one connected component. Therefore,

$$M = 4 - 4 + 2 \times 1 = 2.$$

The cyclomatic complexity is 2.

Higher numbers indicate more possible paths and more complexity. However, many studies have questioned the appropriateness of using the CCM for program comprehension, and it still has some shortcomings [56, 60, 61, 62]. One of the shortcomings is that the branching instructions may not weigh equally. For example, `for` can take twice as long to understand as `if` [61], but the CCM counts both instances as a single

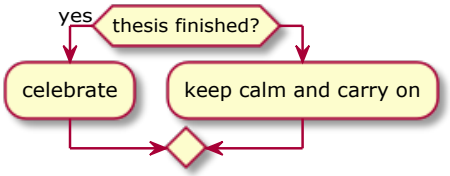


Figure 3.2: Program control flow activity diagram.

node, and another is that the CCM only focuses on the control flow and ignores the data flow, which means that a code block with 200 lines that performs some complex sequential calculation may have the same complexity score as a single line of code.

There is no consistent positive correlation between program comprehensibility and complexity metrics from either the HCM or CCM. Some have argued that, when it comes to indicating program comprehension effort, LOC provides as much insight as the HCM, CCM, or other syntactic complexity metrics that only measure the structure of the codebase or the properties of a statement [19, 60].

In addition, there are attributes in a PR that signify the number of code changes, such as the number of commits and the number of lines changed. The notion of counting is similar to LOC. Moreover, without observing the repository snapshot when a PR was created, those are the only easily obtainable metrics. Since there is no added advantage to using HCM or CCM [19, 60], the LOC metric was linked to those attributes in this study to examine their effects on program comprehension.

3.2.2 *Observing Software Engineers*

The other side of the coin to measuring program comprehensibility is to observe software engineers' cognitive processes. Some researchers have attempted to evaluate the correlation between common software complexity metrics and program comprehensibility [57, 61]. Simultaneously, some studies measured the cognitive process and discussed aspects of a software system that may not be easily quantifiable with the existing metrics. However, observation of software engineers' cognitive processes can be challenging to control due to the human factor involved. Approaches used in previous research have all been imperfect in at least one aspect.

Many researchers set comprehension tasks for software engineers. These tasks present a shorter code snippet, varying from 15 to 200 lines. Some studies asked participants to complete a multiple-choice quiz, and then measured their reaction time

and quiz scores [31, 52, 61]. Similarly, some studies focused on code patterns already acknowledged as prone to misunderstanding, referred to as "atoms of confusion", and asked the participant to evaluate each manually to determine the expected execution output, then compared to a structurally similar code snippet without the atom to exhibit its effect on comprehension [66]. Other studies required participants to complete the blank spaces, and those code snippets were designed to depend on first comprehending the entire snippet [57]. In addition to these conventional studies, which focused on participants' subjective consciousness, new techniques have been emerging. Those techniques combine comprehension tasks and leveraged psycho-physiological sensors to investigate the functions of particular cognitive subprocesses (e.g., language processing, attention, and memory).

For example, Peitek *et al.* [58] used brain imaging and an eye tracker to observe how different brain areas react to defined tasks; this setup is illustrated in Figure 3.3. The study found evidence that program comprehension activates the same area as language processing. Another example is the study by Fritz *et al.* [59], who measured eye tracking, electrical activity of the brain, and electrodermal activity while participants performed defined tasks. The participants ranked the perceived complexity of those tasks, and the study analyzed the relationship between task complexity and the captured biometrics. This approach captured the cognitive process successfully and found a correlation in reaction time among coding convention (e.g., naming variables), different operators (e.g., `+`, `&&`, `>=`), and positive statements versus negative statements (e.g., `if (i == j)` versus `if (i != j)`). However, these code snippets were too short and did not reflect how a software engineer comprehends beyond a single class, a microservice, or a software system. A typical size software system usually contains not only multiple files of source code, but also the navigation mechanism, and it seems that these studies did not consider the required memory bandwidth. Furthermore, multiple-choice is not suitable for the current practical development environment.

Some researchers asked participants to memorize the source code verbatim rather than completing a blank space or answering a multiple-choice quiz. The code included both the sequence flows and values assigned to variables, and participants were asked to transcribe it from memory [30] or shown a slightly modified version and asked to point out the differences [57]. Shneiderman [67], a researcher who studied software psychology and has published several often-cited works since the late 1970s,

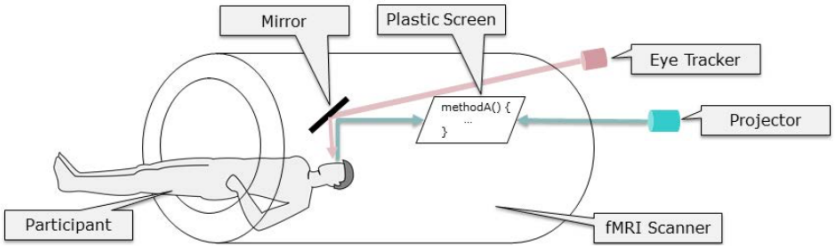


Figure 3.3: Measuring program comprehension with functional magnetic resonance imaging (Reproduced from [58]).

made an interesting analogy between programs and music. He suggested that, if the problem domain is well known to a software engineer and the source code is well structured, then, similar to musicians who can memorize entire symphonies or thousands of music sheets, software engineers should be able to memorize the source code. While evaluating the effect of source code structure on memory could provide valuable insight into the slower reaction of experienced software engineers to unplan-like programs (i.e., violating common programming principles, for example, meaningless variable names or unreachable code), memorization is not necessarily a good indication of comprehensibility, especially since memorizing may be a different cognitive process to comprehending.

Finally, several studies recorded audio or video of participants who think aloud while performing assigned tasks to collect real-time data [51, 53, 68]. A study by von Mayrhauser and Vans [53] used a good-sized codebase with more than 40,000 lines and asked participants to perform bug fixes. Shaft and Vessey [51] worked with professional software engineers, with an average of approximately ten years' experience in the accounting domain, to analyze how they worked with a familiar business domain versus an unfamiliar business domain. These studies captured the exact cognitive process of program comprehension and useful contextual information. Due to the focus on the participants' cognitive process in these studies, however, researchers were able to record only extremely limited information regarding the specific instances when participants were having difficulties in comprehending codebase. In contrast, this current study connects information regarding each instance of comprehension difficulties with the relevant code being reviewed.

RELEVANCE TO THIS STUDY The PR data from GitHub API provides information on attributes related to a PR. Some of that information relates to LOC (i.e., the number

of commits and the number of lines changed). This study examined the related LOC metrics and their effects on program comprehension.

Most of the studies conducted to observe software engineers involved a participant selection process, usually with a small population size, as it is laborious to collect data. There are several pitfalls with the methodologies used in these studies. First, the participant selection process may be subjective to the researchers' own opinion; demographic features, such as years of experience, may not represent actual comprehension ability. Second, the context and format of the questions for comprehension tasks could be misleading. Third, in an observed experiment environment, participants may not display normal behavior or natural cognitive processes accurately.

Many limitations of past studies have been overcome in this study. The data collected from GitHub include more than 1,000,000 code review comments written by software engineers on active projects, with multiple collaborators, for real software projects. There is direct access to the code position for relevant comments, allowing more complex and thorough analysis than in other approaches to observe program comprehension. Furthermore, the use of ML in combination with NLP in the methodology in this study allowed the analysis of a considerably larger number of code review comments.

3.3 TYPES OF PROGRAM COMPREHENSION CHALLENGE

At this point, we had recognized from the program comprehension models that, during a cognitive process, software engineers seek information to understand the codebase. This section describes previous attempts to catalog the specific types of information needs in program comprehension.

A common technique is to conduct qualitative studies on software engineers as they engage in a comprehension task, dissect the communication among software engineers, and categorize the questions that arose [14, 68, 69, 70, 71].

There are four fundamental comprehension interrogative words or particles [68], and most studies expressed the types of information needs in one of these words:

- Why – for example, why is the code implemented this way?
- How – for example, how does the method obtain the result?
- What – for example, what are the variables for the method?

- Whether – for example, whether the method returns the expected result?

While these interrogative words provide an initial direction to identify how possible comprehension challenges can be detected from code review comments, they are too broad, and often other non-comprehension-related questions are expressed in a similar dialogue act.

The types of problems encountered significantly depend on the activity; implementing a feature, reproducing a bug, and code review may each have different problems [14, 70, 72]. Many researchers have formulated more specific lists of questions and categorized them according to different activities. However, some of those questions are too specific to the situation; for example, one of the questions listed by Sillito *et al.* [70] is "What is the correct way to use or access this data structure?"

Therefore, some studies attempted to balance the level of detail in a question. For code-review-related activities, Ko *et al.* [14] formulated a comprehensive list of questions with an adequate level of abstraction. Moreover, a survey of 1,477 respondents conducted by Maalej *et al.* [9] utilized the list of questions from Ko *et al.* [14] and confirmed the relevance of the types of information needs.

3.3.1 *Dialogue Act Classifications*

A dialogue act represents the meaning of an utterance in a dialogue intended by the author. Identifying the dialogue act in a review comment adds to the understanding of the discourse sequence, structure, and interrogative words or particles used [73]. Stolcke *et al.* [73] developed 42 dialogue act classifications based on a corpus of spontaneous human-to-human telephone conversations. These classifications formed the basis of a study done by Wu *et al.* [74] that further refined the number of classifications relevant to online chat conversations to 15.

These dialogue act classifications have been adopted by other studies to compile corpus for NLP and ML [75]; the classifications were: "*Accept*," "*Bye*," "*Clarify*," "*Continuer*," "*Emotion*," "*Emphasis*," "*Greet*," "*Other*," "*Reject*," "*Statement*," "*System*," "*nAnswer*," "*whQuestion*," "*yAnswer*," and "*ynQuestion*."

RELEVANCE TO THIS STUDY Ko *et al.* [14] used a rigorous design to categorize the types of program comprehension challenges. Maalej *et al.* [9] also validated the

same list, and this study adopted a part of the list that is applicable to a code review process:

- What is the program supposed to do?
- What was the developer's intention when writing this code?
- Why was this code implemented this way?
- Who has experience with this code?
- Who wrote this piece of code?

These questions form the basis for defining the rules for the manual labeling of PR review comments. The manually labeled dataset was used to train the ML classifier and the content analysis.

Since GitHub PR review comments are in the form of online chat conversations that exchange communication regarding code changes, the ML training process leveraged the fundamental comprehension interrogative words, using NLP to classify the dialogue act of review comments with the 15 classifications developed by Wu *et al.* [74]. This study discusses the relevance of those dialogue act classifications and program comprehension challenges.

3.4 SUMMARY

The reviewed studies provided important insights into the theories behind cognitive processes and how software engineers seek information. They also provided an understanding of different metrics for measuring program comprehensibility and different approaches to observing how software engineers understand the codebase.

In addition, this review highlighted the need for further research to focus on instances when software engineers experience program comprehension challenges. This study is based on the previous studies and added one more step to overcome many of the limitations. The PR code review data collected include more than 1,000,000 real review comments from real-world software projects. With the implementation of ML and NLP, this study extracted review comments that contained some indication of program comprehension challenges, allowing us to be specific and perform a more thorough analysis than other more general-purpose program comprehension studies.

4

METHODOLOGY

This chapter introduces the research objective, establishes the overall research context, and demonstrates the research design and process in detail. Moreover, it discusses the rationale behind the design decisions and explains how obstacles were mitigated to ensure that the research design and implementation were as rigorous as possible.

4.1 RESEARCH OBJECTIVE

The primary objective of this study is to provide insight into program comprehension challenges during PR code reviews. The intention is to leverage ML to auto-detect instances when a software engineer experiences difficulties in understanding code changes, and, furthermore, to analyze those specific instances and evaluate the types of challenges faced during a PR code review compared to performing a development task. Five main RQs guided the research design:

RQ1. What are the attributes of pull requests that affect program comprehensibility?

- RQ1.a. How does the number of lines of code changed in a pull request affect program comprehensibility?
- RQ1.b. How does the number of commits in a pull request affect program comprehensibility?

RQ2. How accurately can a machine learning classifier detect program comprehension challenges in a pull request code review?

RQ3. What types of program comprehension challenges do reviewers face?

RQ4. Why do reviewers face program comprehension challenges?

First, RQ1 examines measurable attributes that a PR author can control and their effects on program comprehension. RQ1a and RQ1b focus on a subset of attributes that measure the number of code changes. Those attributes are intrinsically similar to the concept of counting in LOC, one of the most common code metrics, which is considered to be a simple yet effective metric to indicate program comprehensibility

[19, 60, 62]. Thus, this study evaluates whether LOC is applicable in the context of code reviews. The answer to this question will guide software engineers and researchers to understand what guidelines can be adopted or which harmful practices to avoid when creating PRs.

Second, RQ2 focuses on the possible implementation of an ML classification model that can automatically identify when a software engineer encounters a problem due to missing knowledge regarding code changes in a PR on an acceptable level.

Third, RQ3 delves into the types of information required during code reviews and highlight the uniqueness of the code review activity compared to other development activities, to spark research interest and context-aware tool development opportunities.

Finally, RQ4 examines the possible factors that prevent software engineers from being able to comprehend the proposed PR code changes.

4.2 METHODOLOGY OVERVIEW

To achieve the research objective, a series of investigations were conducted into code review comments exchanged by software engineers on GitHub when reviewing PRs. The following describes the high-level research process:

1. Data collection
    a) Secondary data of 1,036,743 PR review comments were collected from GitHub and GHTorrent, which comprised both qualitative data, such as the body text of a review comment, and quantitative data, such as the number of lines changed in a PR.
    b) Random samples were selected from the experiment dataset, with a sample size of 770 to provide a 99% confidence level and a 5% confidence interval.
2. Manual labeling of the sample dataset was done in collaboration with independent coders to identify PR review comments where the commenter indicated difficulties in understanding code changes.
3. Feature engineering was performed, and statistical hypothesis tests were conducted to support the discussion for RQ1.
4. A supervised ML classifier was built with the labeled sample dataset and features extracted for detecting program comprehension challenges to answer RQ2.

5. Content analysis
    a) The ML classifier was applied to on the experiment dataset to detect more instances of PR review comments that indicate symptoms of program comprehension challenges, which produced a dataset with a sample size of 384 program comprehension challenges.
    b) Three hundred and eighty-four instances or non-program comprehension challenges were sampled from the experiment sample dataset.
    c) A content analysis was conducted on the two datasets to facilitate the investigations for RQ3 and RQ4.

Before the detailed research procedures are described, it is beneficial to visualize the overall research approach. Figure 4.1 illustrates the high-level research context in scope, presenting the data sources of the PR review comments and how the research process and procedures fitted into the overall setting.

4.2.1 *Resources and Tools*

Python v3.9 applications were implemented in this study for the data collection and scripts used for ML training. The applicable Python packages are listed and described in Appendix A.

The data collection was integrated with GitHub REST API v3 and GHTorrent services, namely BigQuery[1] and MongoDB[2]. GitHub imposed a user-to-server rate limit of 5,000 requests per hour[3]; thus, multiple personal access tokens were required for authentication.

1 GHTorrent on the Google cloud, https://ghtorrent.org/gcloud.html.
2 Collections in MongoDB, https://ghtorrent.org/mongo.html.
3 https://docs.github.com/en/free-pro-team@latest/developers/apps/rate-limits-for-github-apps.

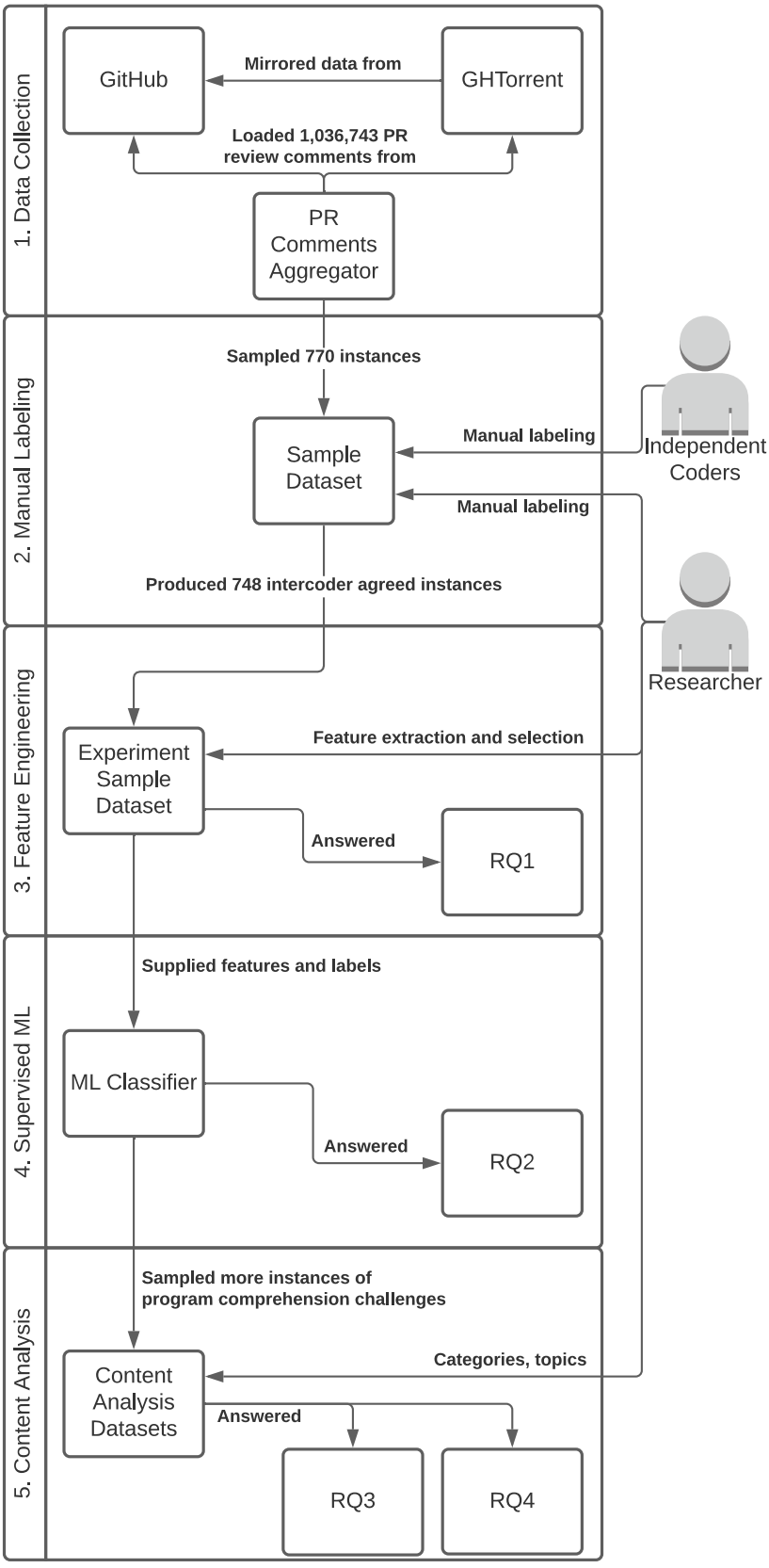


Figure 4.1: Research context diagram. Glossary: machine learning (ML), pull request (PR) and research question (RQ).

4.3 DATA COLLECTION FOR PULL REQUEST REVIEW COMMENTS

Many researchers have utilized comprehension tasks, memorization, or think-aloud protocols to investigate how software engineers understand code, and all involved some form of an interview process. Compared to those methods, major advantages of using PR code review comments were data availability, authenticity, and diversity.

Availability means that the code review comments were written communication and effectively crowdsourced from the reviewers of a popular code hosting service. This method only took a fraction of the time that it would take to conduct interviews to gain a similar amount of information. As a result, this study collected a significantly more extensive dataset.

Authenticity refers to the review comments capturing a more accurate representation of the results of reviewers' understanding, misunderstanding, or inability to understand the code changes as opposed to a lab setting, where participants are conscious of being observed.

Diversity implies that, instead of a carefully crafted code snippet, deliberately manipulated codebase, or a selection of interview participants, the data collected in this study originated from real active open-source projects with a broader mixture of different demographics among reviewers.

In addition, pull-based development is becoming more common [37], and many software engineers will inevitably have to review the work of others at some point during their careers.

4.3.1 *Secondary Data Collection From GitHub and GHTorrent*

GitHub is one of the most popular collaborative code hosting services and has already collected a large number of PR review comments that would have been inaccessible otherwise. Moreover, it offered a comprehensive set of APIs, making it an almost ideal candidate for data mining. However, for the purpose of this study, it lacked an API for querying data of PR review comments across its entire data store without specifying the repository and its owner.

The GHTorrent project built by Gousios [76] addressed such limitations by mirroring the GitHub data. During a PR code review, the author of the PR and the reviewer(s) exchange communication regarding code changes in the form of PR review comments,

and that information is captured in GitHub. Subsequently, GitHub published events, namely, PullRequestReviewCommentEvent.[4] GHTorrent used an event sourcing pattern to listen for those events and persistent-stored the data in various data storages that can be more conveniently queried compared to GitHub. Thus, this study collected and aggregated secondary data from both GitHub and GHTorrent.

The first step in the data collection process was to isolate the domain objects relevant to a PR. Although PR issue comments allow authors or reviewers to communicate and make general comments about the entire PR, they lack specific references to a code change location in a file. Therefore, this study focused on PR review comments and regarded PR issue comments as out of scope.

In total, the PR review comments were collected from 1,969 git repositories, containing 140,986 PRs, with a total of 1,036,743 review comments by 19,336 individual users. Out of those review comments, 286,715 were from the PR author and 750,028 from the reviewers. The collected dataset is summarized in Table 4.1. The following two parts of this section describe in greater detail the selection criteria and collection procedure applied in this study.

# of Repository	# of PR	# of Users	Total # of PR Review Comment	
				1,036,743
1,969	140,986	19,336	**By Author**	**By Reviewer**
			286,715	750,028

Table 4.1: Data collection summary. Glossary: pull request (PR).

4.3.2 *Selection Criteria*

The GHTorrent BigQuery source dataset was produced in April 2018, with a cutoff date of 2018-03-31 for all data previously collected. As a result, the valid date range for the collected data was from February 2011 to March 2018.

Kalliamvakou *et al.* [77] provided some tactical recommendations for mining PR and code review data from GitHub to minimize potential perils. Based on their guidance,

4 https://docs.github.com/en/free-pro-team@latest/developers/webhooks-and-events/github-event-types#pullrequestreviewcommentevent.

the first step of the data collection from GHTorrent BigQuery (see Appendix B) was purposive sampling with the intersection of results filtered by the following criteria:

- Five or more medium-term commits (during the last two years), and five or more recent commits (during the last three months).
- Three or more persons collaborated and implemented changes.
- Five or more medium-term and five or more recently created PRs.
- Prominent programming language was Java.
- Repository was not mirrored from other code hosting services, such as SourceForge, BitBucket.
- Repository was not deleted from GitHub (GHTorrent retains records for deleted repositories).

One of the criteria was selecting Java as the prominent programming language to maintain consistency for the analysis and a finite scope. In addition, Java had consistently been one of the top three popular languages for the last six years, after JavaScript and Python [34]. In addition to the criteria described above, since this study involved content analysis to code the data and develop themes and concepts manually, further filtering was done with a Python application to include only English review comments.

4.3.3 *Data Collection Procedure*

Due to various limitations inherent to each data source, the collection process sourced the data from multiple sources to overcome these limitations. GitHub was the main data source, but its API was not designed for bulk data collection across multiple git repositories. GHTorrent's BigQuery dataset offered a data archive and allowed bulk export of data into a *.csv file. However, there was a length restriction on the review comments stored in the dataset, and comments were truncated after 255 characters. Furthermore, GHTorrent did not capture some crucial PR attributes required for the analysis in this study, so the missing information was restored by retrieving from GitHub.

Figure 4.2 shows the data flow of the collection procedure. First, the PR review comments were produced by PR authors and reviewers on GitHub. Second, this data were subsequently extracted, loaded, and transformed by GHTorrent into its BigQuery and MongoDB format. Finally, to collate the data from the various sources, a Python application named PR Comments Aggregator, [5] was developed. This data aggregator treated the queried result of the PR review comments (as a *.csv file) from BigQuery as an input, processed each comment, and performed further data filtering and enrichment to produce the experiment dataset for this study.

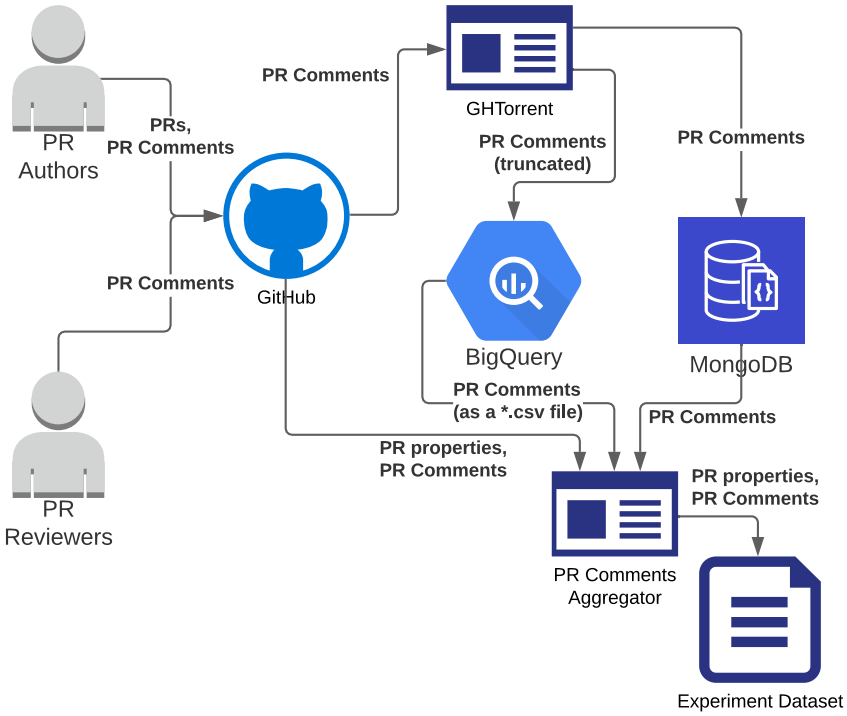


Figure 4.2: Data flow of pull request (PR) review data collection from GHTorrent and GitHub.

Figure 4.3 depicts the data collection activities, further data filtering, and enrichment performed in the PR comments aggregator. First, the data aggregator used a third-party open-source library called cld2-cffi [78] to detect the language used in a review comment and only retained the English ones. Second, by testing whether the text length equals 255, the data aggregator determined whether the review comment had been truncated. If so, the data aggregator expanded it by restoring the missing information from GHTorrent's MongoDB. If the review comment was missing, it was discarded from the experiment dataset output. This scenario could happen if, for

5 Pull Request Comments Aggregator, https://github.com/eddie-chiang/prca.

example, the review comment was deleted on GitHub and the change mirrored to the MongoDB but not in the archived dataset in BigQuery. Finally, the procedure enriched the dataset with more PR attributes that were not captured by GHTorrent by retrieving from GitHub REST API v3. The attributes extracted from each of the three data sources are shown in Table 4.2.

A newer GraphQL[6] API from GitHub was discounted for this study because it was not suitable. The GraphQL API does not have the ability to identify the file associated with a PR review comment. Moreover, there was no overall count of the PR review comments in a PR, which required separate API calls to iterate through the PR review comment lists.

Two obstacles were encountered during the procedure. One was that GHTorrent discontinued access to MongoDB in June 2019 due to the restriction of general data protection regulation (GDPR),[7] and the database contained identifiable personal information, such as email addresses. The bulk of the data collection procedure had been completed at this point, and none of the personal data was within the scope of this study. However, the PR comments aggregator Python application was refactored to revert to GitHub for restoring any truncated PR review comments to ensure resiliency.

Another obstacle was the GitHub API rate limit of 5,000 requests per hour. Three API calls were required for each PR review comment to collect the data required for this study, including obtaining the PR attributes, the PR review comment attributes, and commit file attributes. Three actions were taken to minimize the impact. The first was to seek out peers to donate personal GitHub access tokens to boost the overall limit. As there was a significant amount of data to be collected and APIs to invoke, the second was to parallelize the API calls to minimize the time spent waiting on the I/O. The third was to implement a back-out and cool-down function when all the available rate limits were depleted.

At the end of the data collection procedure, 1,262,684 PR review comments were initially returned from GHTorrent BigQuery. Of these, 162,331 were not in English, and 63,610 could no longer be found on GitHub, resulting in a total of 1,036,743 PR review comments.

6 GitHub GraphQL API, https://docs.github.com/en/free-pro-team@latest/graphql.
7 https://github.com/ghtorrent/ghtorrent.org/pull/696#issuecomment-508579729.

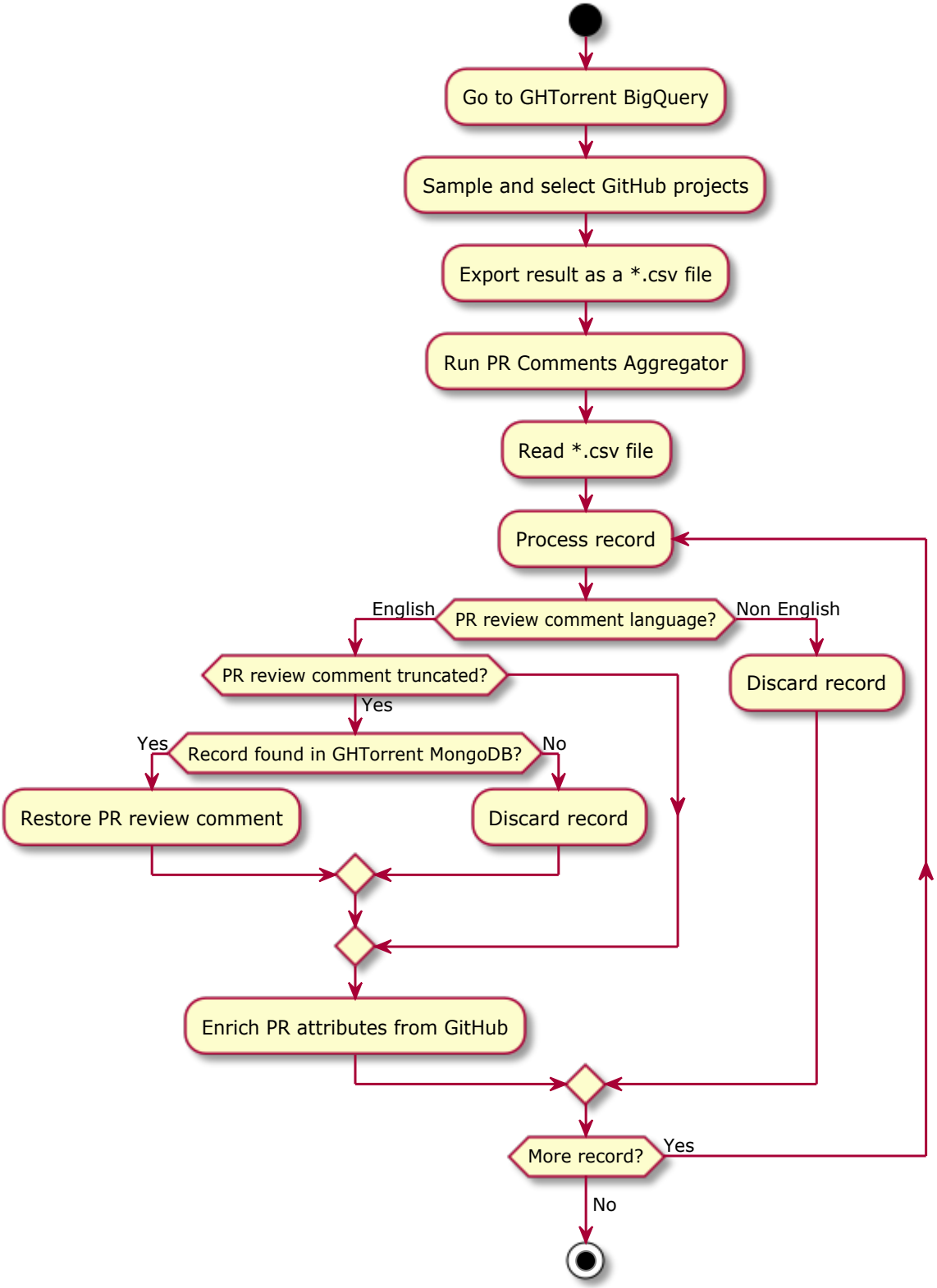


Figure 4.3: Activities of pull request (PR) review data collection from GHTorrent and GitHub.

Property / Attribute	Description	Data Source(s)
Body	The main content of the PR review comment.	1. GHTorrent BigQuery 2. GHTorrent MongoDB 3. GitHub
Position	The position in the code changes where the review comment was added.	GHTorrent BigQuery
Commenter Association	How the commenter associates with the repository. Possible values: MEMBER,CONTRIBUTOR, NONE, COLLABORATOR, and OWNER.	GHTorrent BigQuery
Comment Is by Author	Whether the review comment is from the creator of the PR.	GitHub
PR Issue Comments Count	The total number of PR issue comments in the PR.	GitHub
PR Review Comments Count	The total number of PR review comments in the PR.	GitHub
PR Commits Count	The total number of commits associated with the PR.	GitHub
PR Additions	The total number of lines added from all the commits in the PR.	GitHub
PR Deletions	The total number of lines deleted from all the commits in the PR.	GitHub
PR Changed Files	The total number of files changed from all the commits in the PR.	GitHub
PR Commits Count Prior to Comment	The number of commits associated with the PR just before the review comment.	GitHub
Commit File Status	The file status that the review comment references. Possible values: added, modified, renamed, and removed.	GitHub
Commit File Additions	The number of lines added in the file.	GitHub
Commit File Deletions	The number of lines deleted in the file.	GitHub
Commit File Changes	The number of lines modified in the file.	GitHub

Table 4.2: Attributes collected for pull request (PR) review comments.

4.4 MACHINE LEARNING OF PROGRAM COMPREHENSION CHALLENGES

Machine learning is particularly useful for processing written communication because ML can analyze more data efficiently to detect occurrences when there is an issue with understanding code changes and empowers the study to focus on those specific occurrences.

Moreover, NLP-related applications are increasing, and some provide a comprehensive set of features, both in a commercial application, such as Natural Language API,[8] and open-source projects, such as Natural Language Toolkit (NLTK).[9] Some studies used NLP libraries to perform text analysis and achieved a satisfactory classification performance [46].

Once the PR review comments had been collected, the process of creating an ML classifier was the next stage of the research. Several ML algorithms (also known as learning algorithms) were experimented with to test the manually labeled PR review comments to ascertain whether or not ML could detect a PR review comment indicating a comprehension problem. Given that the goal of the ML model was to classify, supervised learning was applied. Due to the lack of studies on PR review comments, there was no reasonable indication of which NLP preprocessing and ML algorithm would perform best with review comments data. To add to the knowledge, this study investigated four typical NLP preprocessing and six common ML algorithms to do the training with the PR review comments dataset, and utilized quantitative analysis to determine what preprocessing and algorithm performed best on the PR review comments dataset.

Python was selected as the instrument for building the ML classifier, given the maturity and readiness of the third-party libraries available for NLP and ML. The NLTK library provided text corpora for dialogue act classification, word stemming, and lemmatization, and these were employed as part of the NLP preprocessing tasks [44]. Scikit-learn[10] library provided some additional NLP preprocessing functions and the implementation of common ML algorithms [79]. The advantage of scikit-learn over other common libraries, such as TensorFlow [80], was its focus on providing additional functions for feature extraction and a higher level of abstraction of the classifier algorithms. As a result, fewer lines of code were required to consume its APIs. Fur-

8 Natural Language API, https://cloud.google.com/natural-language.
9 NLTK, https://www.nltk.org.
10 scikit-learn: machine learning in Python, https://scikit-learn.org.

thermore, the scikit-learn documentation was well structured and required a smaller learning curve. The objective of this study was to provide a broad understanding of PR review comments data, for which scikit-learn provided a composition mechanism, `Pipeline`, that allowed this study to switch between different NLP preprocessing tasks and classifier algorithms in seconds. Although some degrees of freedom had to be sacrificed to fine-tune the classifier algorithm, combined with some before-mentioned factors, it was an acceptable tradeoff for this study [81].

Figure 4.4 presents an overview of the activities for building the ML classifier for the PR review comments to detect program comprehension challenges. More detailed descriptions of the procedures are provided in the following parts of this section.

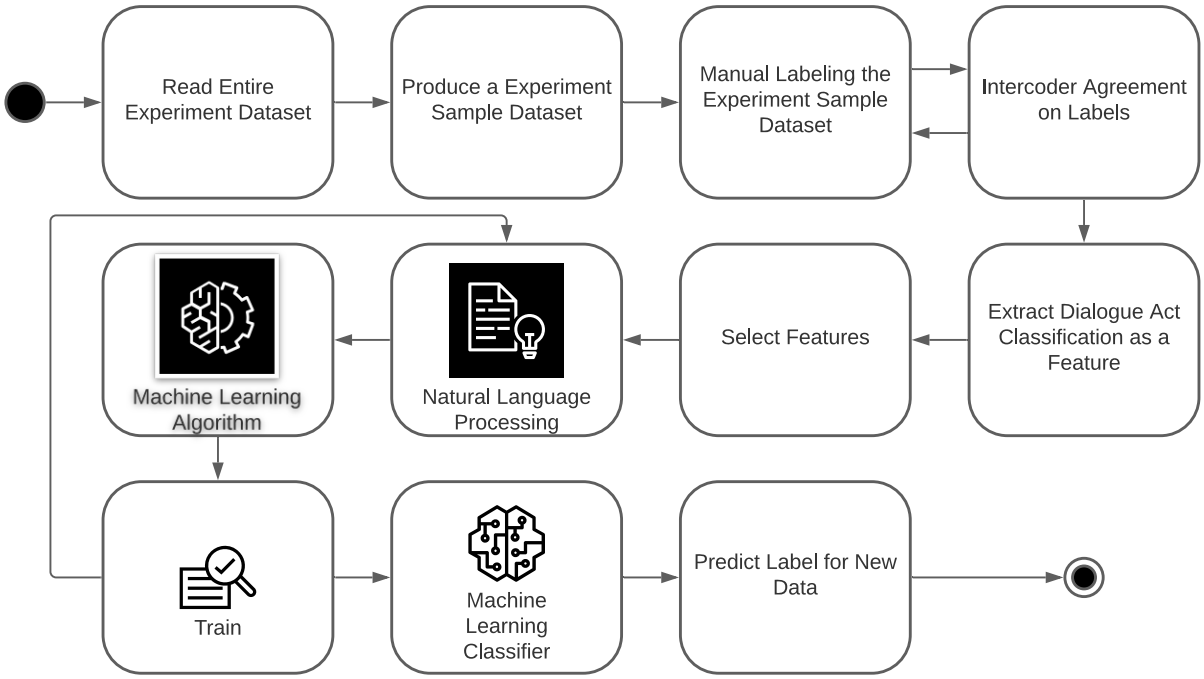


Figure 4.4: Supervised learning activity diagram.

4.4.1 *Data Sampling Procedure for Training and Test Dataset*

To conduct supervised learning, the first step was to create a manually labeled dataset. With more than one million PR review comments collected and given the time and resource constraints, a sampling procedure was employed to obtain a manageable sample of the dataset for the manual labeling task while providing an acceptable level of confidence to reflect the entire content of the collected dataset. The sample size

required was determined with a sample size calculator,[11] and the parameters and result are presented in Table 4.3.

Confidence Level	99%
Confidence Interval	5
Population of PR Review Comments	1,036,743
Sample size needed for manual labeling	665

Table 4.3: Sample size calculation for pull request (PR) review comments manual labeling.

However, to offset potential disagreements from the independent coders during manual labeling, an additional 105 PR review comments were added to the sample to produce a sample dataset size of 770. A Python script was executed to randomly select 770 samples from the experiment dataset, using Python's built-in library, `random`, to produce the sample dataset required for manual labeling.

4.4.2 *Manual Labeling*

Once the sample dataset had been generated, data analysis of the PR review comments and manual labeling were the next steps for supervised learning. These steps were designed to create the training and testing dataset for supervised learning. There are several levels in a code review discussion, including thread, post, and sentence. The data analysis and manual labeling focused on the post level due to the following two reasons:

- While GHTorrent and GitHub provided the ability to retrieve all review comments across all discussion threads in a PR, they lacked an obvious identifier or a straightforward API to group and order PR review comments in a discussion thread.
- With the current limitation in GitHub, a discussion thread could not have a sub-thread; thus, it could contain multiple conversations. For example, two reviewers might comment on the same line of change, each discussing different concerns.

11 Sample Size Calculator, https://www.surveysystem.com/sscalc.htm.

The additional effort required and the potential risk of cross-talk contaminating the data analysis rendered post-level based analysis preferable.

Three independent coders were acquired to perform manual labeling, including two external software engineering Ph.D. students and one professional software engineer with more than 12 years of industry experience. To ensure higher confidence and reliability of the data analysis, the coders independently analyzed the same set of PR review comments with the same labeling scheme. The precedent set by Ko *et al.* [14] and Maalej *et al.* [9] on the types of program comprehension challenges were employed as the basis of the manual labeling guideline.

After the initial round of independent labeling, 453 out of 770 PR review comments had been labeled by all three coders. This sizeable initial set was due to the labeling guideline having been discussed in detail and agreed upon before the labeling process. The initial Krippendorff's alpha measure was 0.6695, calculated with the ReCal[12] utility developed by Freelon [82].

The coders held a recalibration meeting to compare and discuss the label differences and refine the labeling guideline. After the meeting, the second round of independent labeling revisited the previous 453 PR review comments and labeled the remainder of the sample dataset of 770 PR review comments. Another recalibration meeting was held to finalize the labeled data. With a time constraint, the meeting was time-boxed to address as many remaining label discrepancies as time allowed to reach consensus among all three coders. There were 22 remaining discrepancies that were discarded from the dataset. The entire process of data analysis and manual labeling spanned almost six weeks.

As a result of this process, the final Krippendorff's alpha measure of 0.9403 was obtained. The measure was ≥ 0.8, indicating an acceptable intercoder reliability [83]. The details of the intercoder reliability coefficients in the initial round, recalibration round one, and round two are presented in Table 4.4.

A total of 748 PR review comments had agreed labels with consensus from all three coders. The data with the intercoder agreement served as the experiment sample dataset for supervised learning. In the experiment sample dataset, two labels were created for "program comprehension challenge": "*Yes*" and "*No*." Label "*Yes*" indicated that the review commenter expressed some problem related to understanding code

12 ReCal: reliability calculation for the masses, http://dfreelon.org/utils/recalfront.

		Recalibration	
	Initial Round	**Round One**	**Round Two**
Number of Coders	3	3	3
Number of Cases	453	770	770
Average Pairwise Percent Agreement	90.29%	96.5%	98.2%
Pairwise Agreement Coders 1 & 3	92.27%	96.5%	98.2%
Pairwise Agreement Coders 1 & 2	91.39%	97.7%	99%
Pairwise Agreement Coders 2 & 3	87.2%	95.2%	97.4%
Krippendorff's alpha	0.6695	0.8827	0.9403

Table 4.4: Intercoder reliability coefficients.

changes, while label "*No*" indicated that the review commenter did not experience any program comprehension issue.

There were 131 instances of program comprehension challenges in the 748 PR review comments, versus 617 instances that were not challenge related. Table 4.5 provides examples of the labeled instances.

Label	**Type of Information Need**	**Example**
Yes	What is the program supposed to do?	how is this used?
Yes	What was the developer's intention when writing this code?	Which error are you trying to defend against here?
Yes	Who has experience with this code?	How do I sort a single 'pom.xml' file?
Yes	Why was this code implemented this way?	Shouldn't it be done when we create topic?
No	-	Missing indentation.

Table 4.5: List of program comprehension challenge labels and examples.

There were five instances noted and discussed during the manual labeling where review comments did not provide enough context. One example was "Can we use isNotBlank()?" These scenarios were caused by the data collection limitation, which prevented the retrieval of the entire discussion thread for the review comments.

Furthermore, several review comments were difficult to distinguish if the commenter mentioned a program comprehension challenge or made a polite, indirect, soft suggestion. The scenarios where the comment contained a concrete implementation suggestion were treated as soft suggestions, indicating that the commenter understood the code, such as "add a message as the second argument?" Figure 4.5 illustrates the labeling guideline.

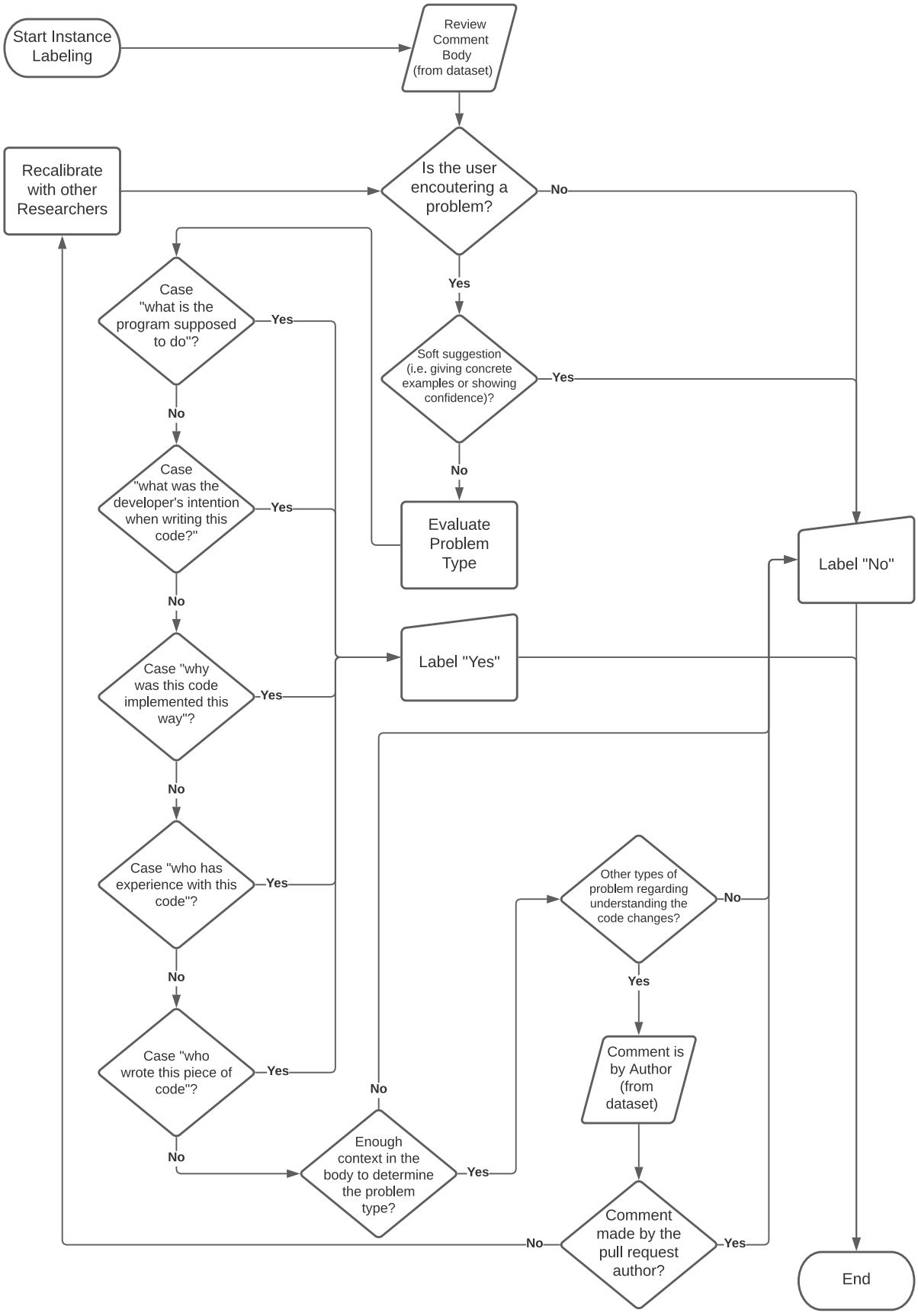


Figure 4.5: Program comprehension challenge labeling guideline.

4.4.3 *Feature Engineering*

The feature extraction, selection, and NLP preprocessing procedures and the features obtained for ML are described in this subsection.

4.4.3.1 *Feature Extraction for Dialogue Act*

Feature extraction from text data is a technique in feature engineering. Its goal is to derive informative values for ML algorithms [84]. A PR review comment discussion is fundamentally a conversation between the PR author and reviewers. This study sought to apply an ML classifier for NLP and extract the dialogue act from the PR review comments to identify the interrogative words or particles that often appeared when information needs were discussed [68].

A dialogue act classifier was built, based on the algorithm from Bird *et al.* [44] that demonstrated the capability to identify dialogue act using the NLTK library. The algorithm utilized the Naval Postgraduate School (NPS) chat corpus[13] in NLTK, which contained more than 10,000 posts from online chat services. Forsythand and Martell [75] labeled the entire corpus with the 15 dialogue act classifications developed by Wu *et al.* [74]. Table 4.6 lists all 15 classifications and example posts for each classification.

The procedure included a preprocessing step to tokenize words in the posts to extract as features and the naive Bayes algorithm was selected to build the classifier with a train-test split of 90% training and 10% test. Even though precision and recall rates for some of the dialogue act classifications were low, this study concentrated on identifying PR review comments with interrogative words or particles. Among the dialog act types, "*whQuestion*" and "*ynQuestion*" contained the use of interrogative words or particles. Thus, these were the focal points for performance evaluation.

For "*whQuestion*," the classifier performed at a precision rate of 74.6% and a recall rate of 76.9%. For "*ynQuestion*," the precision rate was 67.1%, and the recall rate 73.4%. Precision and recall are the metrics for evaluating the accuracy of the detection of an ML classifier. The definitions of these metrics are expanded on in Section 4.4.5. In simple terms, a higher percentage indicates more accurate detections than a lower percentage, and these rates were acceptable for this study. The detailed performance results for the classifier are presented in Chapter 5.

13 The NPS Chat Corpus, http://faculty.nps.edu/cmartell/NPSChat.htm.

Dialog Act	Example
Accept	ok, victory is yours.
Bye	cya later guys
Clarify	i mean the pepper steak lol
Continuer	Or 5 times.
Emotion	HAHAHA
Emphasis	i love jesus more than ANYONE ELSE
Greet	hey
Other	o
Reject	Test not complete
Statement	that sounds painful
System	JOIN
nAnswer	not that I was aware of
whQuestion	Why not use the client that's already created for the integration tests?
yAnswer	Yes, I'll use a regex
ynQuestion	Does this have to be public?

Table 4.6: Dialogue act classifications and examples.

Subsequently, the classifier labeled the entire experiment sample dataset and added a column, "dialogue act classification." This feature evidently showed statistical significance in relation to program comprehension challenges, which are discussed in the next part.

4.4.3.2 *Feature Selection Using Statistical Hypothesis Testing*

The ML algorithms use input data to perform detections, and the input data are a set of features. The appropriate features are the primary information gain to facilitate building high-performing ML classifiers [84]. The experiment sample dataset was inspected and statistical hypothesis testing applied to establish the PR attributes that had any bearing on whether a commenter had program comprehension challenges.

Chi-squared tests of independence [85] were performed to determine the correlation between the label "program comprehension challenge" and each categorical attribute. In contrast, Mann-Whitney U tests [86] were done to compare the differences in the distributions of each discrete attribute.

These statistical hypothesis tests revealed a significant relationship between the attribute "comment is by author" and the label "program comprehension challenge." Reviewer comments were more likely to express program comprehension challenges

than comments from authors. Even though the relationship may seem intuitive, ten instances were observed where the author had encountered comprehension challenges for the PR that they had created. Likewise, the relationship between "dialogue act classification" and "program comprehension challenge" was significant.

On the contrary, the tests revealed no significant association between "program comprehension challenge" and other attributes. At the end of the feature selection procedure, "comment is by author" and "dialogue act classification" were the only relevant attributes in addition to the main body of PR review comments.

Property / Attribute	Type	Code Metrics?	Example	Selected?
PR Commits Count	Discrete	Yes	17	No
PR Additions	Discrete	Yes	949	No
PR Deletions	Discrete	Yes	485	No
PR Changed Files	Discrete	Yes	7	No
Commit File Status	Categorical	Yes	modified	No
Commit File Additions	Discrete	Yes	1	No
Commit File Deletions	Discrete	Yes	1	No
Commit File Changes	Discrete	Yes	2	No
Commenter Association	Categorical	No	CONTRIBUTOR	No
Comment Is by Author	Categorical	No	FALSE	Yes
PR Issue Comments Count	Discrete	No	0	No
PR Review Comments Count	Discrete	No	55	No
PR Commits Count Prior to Comment	Discrete	No	14	No
Dialogue Act Classification	Categorical	No	whQuestion	Yes

Table 4.7: Feature selection of pull request (PR) attributes.

Table 4.7 lists the outcome of the feature selections. The results are further expanded and elaborated on in the discussion of RQ1 and RQ2 in Chapter 5.

4.4.3.3 *Preprocessing of Pull Request Review Comments*

At this stage, the data preparation procedures had been completed. Figure 4.6 depicts how the program comprehension challenge labels from the intercoder agreement and the dialogue act classification were associated with the experiment sample dataset and split into a training dataset and a test dataset. The subsequent procedure commenced preprocessing the body text PR review comments in the training dataset.

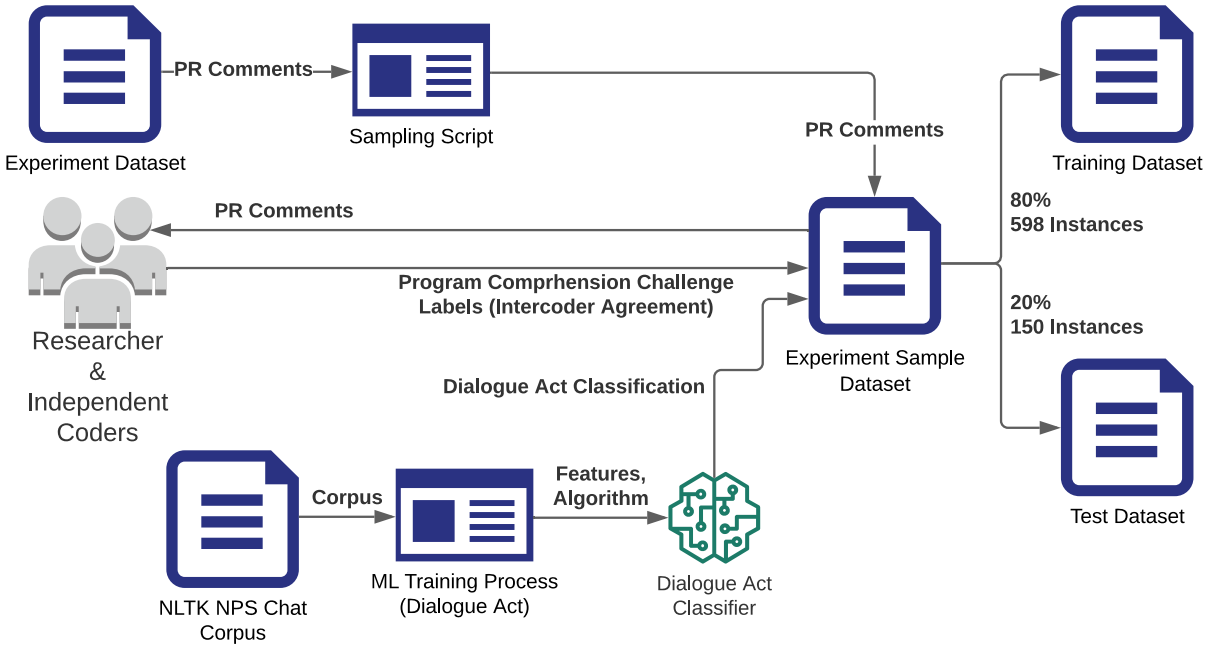


Figure 4.6: Data flow of the preprocessing. Glossary: machine learning (ML), Natural Language Toolkit (NLTK), Naval Postgraduate School (NPS), and pull request (PR).

Although the body texts of PR review comments are in natural language, a NLP step was necessary to transform the body text into features and numeric representations that would be meaningful and ingestible to an ML algorithm. Hence, NLP was applied to the body text during the preprocessing in a further step of feature engineering to extract features and filter inappropriate features to reduce any noise that could adversely impact the algorithm [44, 84]. The following body text is used as an example to clearly explain the set of NLP tasks applied:

> *Without this patch applied, I see that line about 40 times. I also do not understand how these properties could affect that line?*

The first three tasks were feature filtering related to the removal of features that were unlikely to be useful for the classifier. The first task was normalizing the text into all lowercase so that differences between "Without" and "without" can be ignored: "*w*ithout this patch applied, *i* see that line about 40 times. *i* also do not understand how these properties could affect that line?" The second task was removing stop words (e.g., "this," "that"). Silva and Ribeiro [87] suggested that this task tends to improve the classifier performance, which was also evident in this study. This task utilized the built-in stop list word from scikit-learn to weed out pointless features occupied by those common

words: "*patch applied, line 40 times. understand properties affect line?*" The third task utilized the WordNet corpus in NLTK to lemmatize words to group various forms of a dictionary word into one, such as the suffix was removed from "times" and, thus, it became "time": "*patch applied, line 40* ***time****. understand* ***property*** *affect line?*"

The two remaining tasks were feature extraction related. The bag-of-n-gram task involved counting the frequency and possible meaningful collocation of words (e.g., "patch applied"). For instance, if the word "line" appeared more frequently in the body text with program comprehension challenge label "*Yes*", then the classifier may have a higher probability of detecting other body text containing the same word as "*Yes*" [84]. Bag-of-n-gram with one to two words follows:

```
{'patch': 1, 'applied': 1, ',': 1, 'line': 2, '40': 1, 'time': 1, '.': 1,
 ↪ 'understand': 1, 'property': 1, 'affect': 1, '?': 1, 'patch applied': 1, 'applied
 ↪ ,': 1, ', line': 1, 'line 40': 1, '40 time': 1, 'time .': 1, '. understand': 1,
 ↪ 'understand property': 1, 'property affect': 1, 'affect line': 1, 'line ?': 1}
```

The final NLP task was term frequency-inverse document frequency (tf-idf), which was a simple yet effective transformation of the results from bag-of-n-gram. Their relationship can be expressed through the following equation [79, 84]:

$$\text{tf-idf}(t,d) = \text{bag-of-n-gram}(t,d) \times (\log \frac{1+n}{1+\text{df}(t)} + 1),$$

where bag-of-n-gram(t,d) is the number of times a term t appeared in body text d, n is the total number of body text in the training dataset, and df(t) is the number of body text in the training dataset in which term t appeared. With the count from bag-of-n-gram, tf-idf computed a normalized weighting that considered the frequency in which a term appeared across the entire body text in the training dataset. If a word or phrase appeared in many PR review comments, it may be less meaningful and less influential for classifier detection and resulted in a lower weighting [84]. An example of this would be that, if all PR review comments used the word "affect", the classifier would effectively ignore the word as a feature.

In comparison, this study also experimented with other combinations of NLP tasks. One of those was bag-of-words, a simpler variation of bag-of-n-gram that counted the number of times a single word appeared in the body text. The other NLP tasks attempted to retain stop words and various forms of dictionary words, in other words, no lemmatization. Finally, applied stemming in place of lemmatization removed any suffixes from words, for example, "properties" would be lemmatized as

"property" while stemmed as "properti." However, those experimental procedures described in the previous paragraph did not improve performance compared to the previously described procedures. The results and performance comparison is presented in Chapter 5.

This subsection described the NLP of the PR review comments body texts for feature filtering and feature extraction. The next subsection describes how the extracted features from the body text and previously selected features "comment is by author" and "dialogue act classification" were used to build the ML classifier.

4.4.4 *Machine Learning Classifier Training*

This part of this thesis describes in greater detail the ML classifier training procedure to build an ML classifier. The research objective of this study was to assess whether the ML classifier can detect program comprehension challenges and evaluate different feature preprocessing tasks and ML algorithms. The overall data flow and classifier training context are illustrated in Figure 4.7.

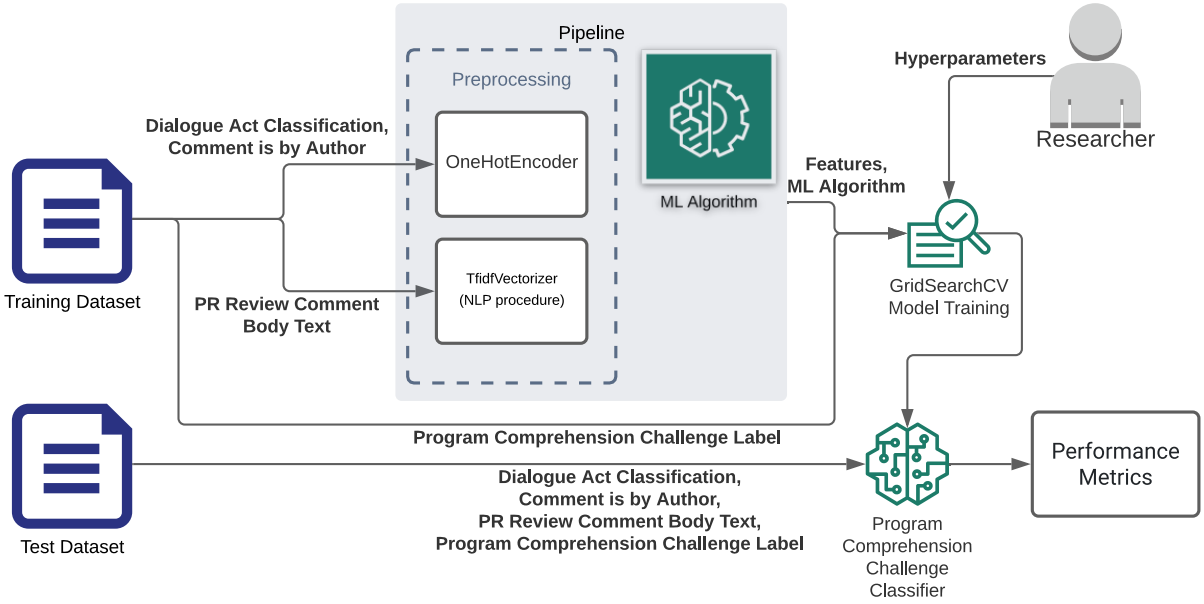


Figure 4.7: Data flow diagram of the classifier training with scikit-learn. Glossary: machine learning (ML) and pull request (PR).

A Python script using the scikit-learn library and its `Pipeline` composition interface was implemented in this procedure, which allowed the orchestration of different

combinations of feature preprocessing tasks and ML algorithms by simply changing the experiment parameters [81]. The `Pipeline` was composed of two main parameters:

1. Preprocessing tasks to transform features into numeric representations.
2. An ML algorithm.

4.4.4.1 *Preprocessing Tasks*

The feature preprocessing tasks involved `OneHotEncoder` and `TfidfVectorizer`. The `TfidfVectorizer` was for the NLP tasks to filter and extract features from the body text, which has already been described and explained in Section 4.4.3.3. In addition, `OneHotEncoder` was utilized to transform the features "dialogue act classification" and "comment is by author" into numeric representations. For example, "comment is by author" was a feature with two possible categories: true or false. `OneHotEncoder` created a binary column for each category, and the transformation is briefly illustrated in Table 4.8.

Row	Comment Is by Author
1	true
2	false

(a) Before transformation.

$\longrightarrow$

Row	Category "true"	Category "false"
1	1	0
2	0	1

(b) After transformation.

Table 4.8: An example of OneHotEncoder transforming a feature into a numeric representation.

4.4.4.2 *Machine Learning Algorithms*

This study evaluated three well-known ML algorithms that have displayed satisfactory text classification performances. The process was repeated with different compositions of preprocessing tasks, ML algorithms, and hyperparameters to assess whether an ML classifier can detect program comprehension challenges in PR review comments and evaluate the performance of each of the compositions.

MULTINOMIAL NAIVE BAYES The first ML algorithm was multinomial naive Bayes. The name "naive Bayes" comes from the algorithm's assumption of treating each feature in the dataset independently given the context of classification, despite that

most of the use cases in reality often display some correlation among the features [88]. However, several studies have found acceptable performance from the application of text categorization [89, 90]. This study aimed to assess the performance of this algorithm with PR review comments, as the set of selected features from the experiment dataset had not been studied previously, according to the literature review.

LOGISTIC REGRESSION The classes for "program comprehension challenge" were a binary set of "*Yes*" and "*No*." The experiment sample dataset proved to be an imbalanced dataset, with more than four times more instances of "*No*" than "*Yes*". Logistic regression is a linear statistical model that aims to find a hyperplane that approximately separates the numeric vector representations of the PR review comments features into two desired classes [90]. Some studies have found logistic regression effective on imbalanced datasets [91]. This study examines the logistic regression algorithm using the imbalanced labeling of PR review comments.

LINEAR SUPPORT VECTOR MACHINE WITH STOCHASTIC GRADIENT DESCENT LEARNING Lastly, the third ML algorithm was a linear support vector machine (SVM) with stochastic gradient descent (SGD) learning. Linear SVM is closely related to logistic regression, further evaluates the margins of hyperplanes to the vectors of each class, and finds the hyperplane with the maximum margin. In other words, it aims to find the most clean-cut way to separate the vectors from two classes [90]. SGD is an add-on to the algorithm; it is an optimization technique for fitting the training data efficiently [79]. Many studies have found that SVM performs better than other algorithms during text categorization [90, 92].

4.4.4.3 *Cross-Validation*

Once the preprocessing tasks and ML algorithms had been chosen and assembled as a `Pipeline`, the `Pipeline` was passed into `GridSearchCV` for fitting with the training data and cross-validation was performed to evaluate different hyperparameters of the preprocessing tasks and the ML algorithms.

Listing 4.1 exhibits a code snippet that exemplifies the options of hyperparameters as `param_grid` being evaluated for the processing tasks, including whether to lemmatize the body text and filter stop words, for the number of words to use for bag-of-n-gram, and the weightings for individual features that could affect the influence of a feature.

Those options were passed into `GridSearchCV` that performed 5-fold cross-validation on the training dataset.

Listing 4.1: Cross-validation to evaluate different hyperparameters using GridSearchCV.

```
pipe = Pipeline(steps=[
        ('preprocessor', preprocessing_tasks),
        ('alg', SGDClassifier())])
param_grid = [{
        'preprocessor__body__tokenizer': [None, LemmaTokenizer()],
        'preprocessor__body__stop_words': [None, 'english'],
        'preprocessor__body__ngram_range': [(1, 1), (1, 2), (1, 3), (2, 3)],
        'preprocessor__transformer_weights': [
                {'body': 4, 'dac': 1, 'is_author': 2},
                {'body': 1, 'dac': 1, 'is_author': 1}],
        'alg__random_state': list(range(1, 20))
    }]
clf_grid_search_cv = GridSearchCV(pipe, param_grid=param_grid, cv=5)
clf_grid_search_cv.fit(X_train, y_train)
```

This study involved manual data labeling of a sample size of 748, which was not ample to waste and could take advantage of cross-validation to sacrifice computation time spent on multiple training iterations, over a smaller sample size [79]. The 5-fold cross-validation split a training dataset into five stratified sets, preserving the percentage of instances for each label, and performed five iterations of classifier training. Each iteration fit four sets of training data to the classifier and retained one of those five sets to evaluate the classifier performance. The results of the evaluation of the five iterations were used to determine the fine-tuned parameters. The final evaluation to measure the trained classifier's performance was validated against the test dataset, which was held-out and invisible to the classifier. Figure 4.8 is a good illustration of how the datasets breakdown was conducted during the cross-validation.

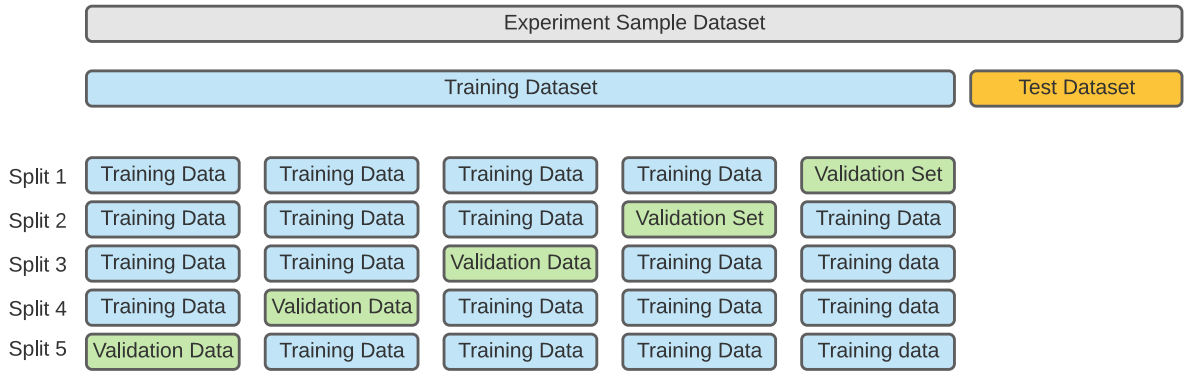


Figure 4.8: Experiment datasets breakdown for 5-fold cross-validation.

4.4.4.4 *Other Machine Learning Training Techniques Considered*

This study explored other ML classifier training techniques, including active learning and evenly distributing the imbalanced dataset due to the smaller sample size.

Active learning involves the sampling of more unlabeled instances, manually labeling them, and adding the training dataset to retrain the classifier. This study attempted to utilize an active learning scenario with pool-based sampling to select unlabeled PR review comments from the experiment dataset. The selection criteria were based on the query strategy with the least confidence based on the ML classifier's detection probability. However, the ML classifier performance did not improve after 30 iterations with a total of 30 new samples.

The other technique was to balance the percentage of samples from different classes in the training dataset, so that the number of instances labeled "*Yes*" versus "*No*" were the same. The balanced training dataset was then given to the ML algorithms to build the classifiers. Evenly distributing the dataset allowed some compositions of preprocessing tasks and ML algorithms to perform with higher accuracy. However, the ML classifier with the best performance did not rely on the evenly distributed training dataset, and its model was trained from the imbalanced training dataset.

Since those two techniques did not improve the classifier's performance over the previously described methodology, they were not deemed suitable for the research objective.

4.4.5 *Machine Learning Classifier Evaluation*

The three selected algorithms were all utilized to fit the training dataset to build the ML classifier and evaluated with the test dataset to compare the classification performance and answer RQ2. The research objective was to identify occurrences of program comprehension challenges and conduct a content analysis on those occurrences.

Therefore, to take advantage of the ML classifier to sample those instances from the experiment dataset, the deterministic metric for comparing the ML classifier performance was based on the F-beta score with $\beta = 0.5$ to favor precision over recall, to have as many correctly detected program comprehension challenge instances as possible, and also to minimize those instances that slip through the detection.

Precision measures the percentage of classified instances that have the correct relevant label, that is

$$precision = \frac{tp}{tp + fp},$$

and *recall* measures the percentage of instances of a relevant label that has been classified correctly, that is

$$recall = \frac{tp}{tp + fn},$$

where tp is the number of true positives, fp is the number of false positives, and fn is the number of false negative [93]. F-beta balances between precision and recall for a more harmonic detection [94, 95], and is calculated as follows

$$F_\beta = (1 + \beta^2) \times \frac{precision \times recall}{\beta^2 \times precision + recall},$$

where lower β puts more weight on the *precision* score, and higher β puts more weight on the *recall* score. This study chose $\beta = 0.5$ to favor precision while balancing an acceptable recall score.

Figure 4.9 portrays the relationship between correctly classified instances (i.e., true positives) and misclassified instances (i.e., false positives and false negatives) that affects the precision and recall measures, and, in turn, affects the F-beta score.

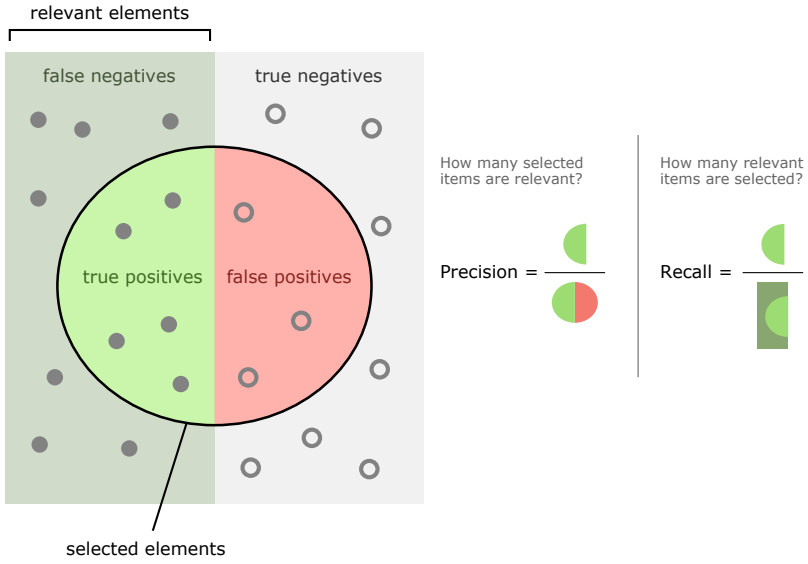


Figure 4.9: Relationship between precision and recall (Adapted from "Precision and recall" by Walber, licensed under Creative Commons Attribution-Share Alike 4.0 International: https://creativecommons.org/licenses/by-sa/4.0).

4.5 CONTENT ANALYSIS OF PROGRAM COMPREHENSION CHALLENGES

Content analysis is one of the most common techniques for qualitative analysis. This involved large amounts of textual data. An ML classifier was used for the initial filtering of data to select only the occurrences of program comprehension challenges, as set out in the research objective. When investigating these occurrences, the applicable categories of the types of program comprehension challenges deduced by Ko *et al.* [14] and Maalej *et al.* [9] were implemented. Therefore, content analysis would be more useful for identifying and characterizing the subcategories of the labeled data effectively and reliably, compared to a thematic analysis method [83, 96]. Moreover, this method enabled inferences about the reviewers.

To gain more insight into the types of program comprehension challenges the reviewers face and the reasons for these occurrences, a manual content analysis was conducted with 384 samples of program comprehension challenges from the PR review comments in the experiment dataset collected from GitHub and GHTorrent, as described in Section 4.3.3. A program comprehension challenge in PR review comments was defined as a commenter expressing information needs, and the guideline for

identifying those instances is illustrated in Figure 4.5. The overall data flow of the data sampling and context of the content analysis are depicted in Figure 4.10.

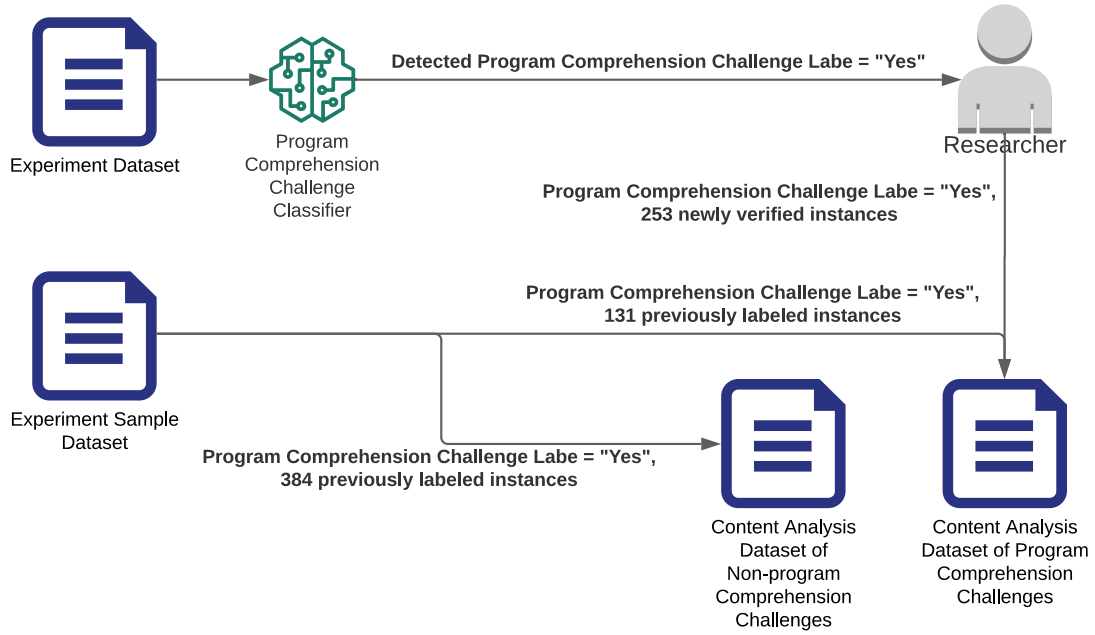


Figure 4.10: Data flow and data sampling for the content analysis.

4.5.1 *Data Sampling Procedure for Content Analysis Dataset*

The manual content analysis samples comprised the already manually labeled 131 instances from the experiment sample dataset, and a new set of 253 randomly selected instances from the pool of experiment dataset. There were 1,036,743 PR review comments in the experiment dataset, and a sample size of 384 would provide a confidence level of 95% with a confidence interval of 5%.

The developed ML classifier was applied to detect program comprehension challenges during the random sampling procedure. Since the precision of the ML classifier was not perfect, a manual labeling procedure was required to confirm the detected samples. Irrelevant PR review comments were rejected and new samples sourced from the pool to replace those. This process was repeated several times to remove the false positives to generate a new set of 253 samples.

Once 253 new instances had been carefully and precisely sampled, they were combined with the 131 previously labeled instances to create the content analysis dataset with a total of 384 samples.

4.5.2 *Meaning Unit, Topics, and Categories*

The meaning unit was defined in this study as the body of text of a PR review comment and its relevant conversation. The content analysis dataset also contained the web page link to the discussion thread on GitHub for each one of the samples; therefore, the meaning unit included the related discussion in the thread in addition to the review comment.

These meaning units were condensed into a higher level of abstraction, formulated into a set of topics. A topic could be considered as a code or a label, usually one or two words in length, that described the core meaning of either an entire meaning unit or a part of it. To cater for possible different comprehension models reviewers might have applied during the code review, the topics were based on the three fundamental knowledge structures described by von Mayrhauser and Vans [53] in the integrated program comprehension model, namely program domain knowledge, problem domain knowledge, programming plans, and rules of discourse.

The classification of the types of information needs by Ko *et al.* [14] have been used in other studies to investigate intrinsic properties of program comprehension activities [9, 14]. This study adopted those types of information needs to define the set of categories. Subsequently, a set of five categories were created. The list of rules for assigning a meaning unit to a category is presented in Table 4.9.

Category	Rules for Category Assignment
Intended program behavior	What is the program supposed to do? Inquiring about the execution outcome, use cases, or expected input.
Developer's intention	What was the developer's intention when writing this code? The effect of the code change is unclear, questioning the reason for the change.
Design rationale for the implementation	Why was this code implemented this way? The intention and effect of the code change are understood, but the reason for the particular implementation approach is unclear, given there may be other options.
Developer familiar with the code	Who has experience with this code? Asking others about how to achieve an implementation goal.
Author of the code	Who wrote this piece of code? Asking who is the author of the code change.

Table 4.9: List of categories and rules for category assignment.

4.5.3 *Manual Labeling*

Once the meaning unit, the rules for categorizing a topic, and the set of categories had been established, the next step of content analysis was to conduct data analysis and manual labeling of the dataset. Each PR review comment in the dataset was assigned an appropriate category and the keywords of the topics that condensed the core meaning were recorded.

Identifying and condensing the meaning unit into topics was a continuous process of refinement by iteratively working on the dataset. For example, a previously recorded topic was tweaked to reflect the core meaning more closely after analyzing several meaning units with similar themes.

The finalized content analysis contained a refined list of 94 distinct topics that captured the core meaning of the PR review comments from the 384 samples, and each meaning unit was also assigned to a category of the relevant type of information need. The outcome of this analysis facilitated the discussion of RQ3.

4.5.4 *Analyzing Non-Program Comprehension Challenges*

Furthermore, to supplement and compare the content analysis for program comprehension challenges, an additional content analysis was conducted on 384 samples of non-program comprehension challenges related PR review comments. The experiment sample dataset already contained more than 600 instances of non-program comprehension challenge related PR review comments, so the content analysis sampled directly from the dataset and randomly selected 384 samples. The same definition of meaning unit and the principle of generating topic discussed in Section 4.5.2 were applied.

Ninety-six distinct topics were derived from the content analysis for non-program comprehension challenge review comments. The comparison between program comprehension challenge versus non-program comprehension challenge review comments was designed to investigate why the reviewers face program comprehension challenges, as discussed in RQ4 of the research objective.

5

RESULTS

This chapter presents qualitative and quantitative evidence for each RQ regarding program comprehension challenges during code reviews, based on the experiment dataset that contained 1,036,743 PR review comments collected from GitHub.

5.1 PULL REQUEST ATTRIBUTES AFFECTING PROGRAM COMPREHENSIBILITY

The first question in this study sought to determine the quantifiable PR attributes and evaluate the associated program comprehensibility consequences:

> *RQ1. What are the attributes of pull requests that affect program comprehensibility?*

The evaluation of the associated program comprehensibility consequences was done on a subset of the experiment dataset, with 748 randomly sampled PR review comments. The experiment sample dataset included 572 unique users, 740 individual file commits, 736 unique PRs, and covered 326 unique code repositories.

5.1.1 *Attributes with Significant Correlations*

For the following results, H_0 denotes the null hypotheses, H_1 denotes the alternate hypotheses, and the level of significance for those statistical hypothesis tests was $\alpha = .05$. The detailed chi-squared test [85] results with the number of actual instances versus expected instances are presented in Table 5.1.

For the PR attribute, "comment is by author," the hypotheses were:

H_0: Being a PR author has no effect on the likelihood of encountering program comprehension challenges.

H_1: Being a PR author affects the likelihood of encountering program comprehension challenges.

	Program Comprehension Challenge						
	Yes		No		Actual		
Pull Request Attribute	**Actual**	**Expected**	**Actual**	**Expected**	**Total**	X^2	***p*-value**
Comment Is by Author						32.529	< .05
FALSE	121	94.4	418	444.6	539		
TRUE	10	36.6	199	172.4	209		
Dialogue Act Classification						103.648	< .05
Accept	0	0.4	2	1.6	2		
Clarify	49	59.0	288	278.0	337		
Continuer	0	0.7	4	3.3	4		
Emotion	0	0.4	2	1.6	2		
Emphasis	5	2.8	11	13.2	16		
nAnswer	9	12.3	61	57.7	70		
Other	10	7.0	30	33.0	40		
Reject	8	8.6	41	40.4	49		
Statement	5	22.6	124	106.4	129		
System	0	0.7	4	3.3	4		
whQuestion	24	7.2	17	33.8	41		
yAnswer	0	1.8	10	8.2	10		
ynQuestion	21	7.7	23	36.3	44		

Table 5.1: Chi-squared test results of machine learning features. N.B. $\alpha = .05$.

A chi-squared test showed that there was a significant correlation between "comment is by author" and the label "program comprehension challenge," consistent with H_1, $X^2(1, N = 748) = 32.529, p < .05$.

It was apparent from the result that PR authors encountered few program comprehension challenges. The number of expected instances was 36.6, which is significantly higher than the actual observed 10 instances. The number of expected instances among reviewers was 94.4, which was significantly lower than the actual number of 121.

The overall confusion matrix for the PR attribute "dialogue act classification" extracted from the PR review comment by the ML classifier is illustrated in Figure 5.1.

The hypotheses for "dialogue act classification" were:

H_0: The dialogue act expressed in a PR review comment has no correlation with program comprehension challenges.

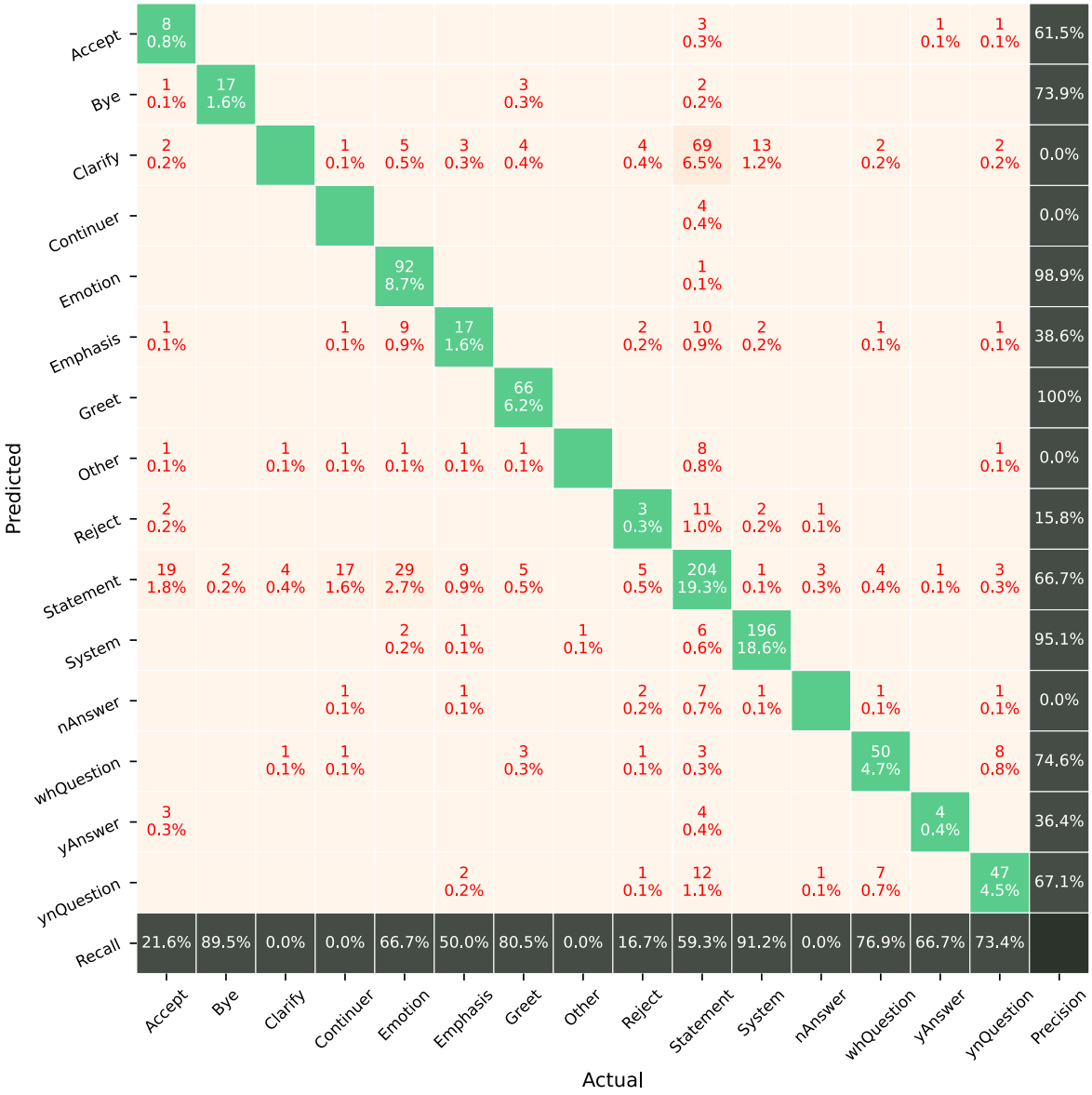


Figure 5.1: Confusion matrix for the dialogue act classification.

H_1: The dialogue act expressed in a PR review comment correlates to program comprehension challenges.

The correlation between "dialogue act classification" and "program comprehension challenge" was again found to be significant, with $X^2(13, N = 748) = 103.648, p < .05$, consistent with H_1.

Among the 748 samples, "*emphasis*" was found in two instances and "*other*" in three instances more than expected. Furthermore, the most striking result to emerge from

the data for "*whQuestion*" that was found to be considerably higher than expected, by more than 16 instances, and for "*ynQuestion*," 13 actual instances more than expected.

5.1.2 *Other Attributes Not Affecting Program Comprehensibility*

On the contrary, the statistical hypothesis test found no relationships for all of the other PR attributes collected in the data. A series of chi-squared tests were conducted for the categorical PR attributes. Similarly, a series of Mann-Whitney *U* tests [86] were done for the discrete PR attributes. The samples were divided into two groups:

1. Program Comprehension Challenge = Yes (GP-Y) – PR review comments indicated program comprehension challenges.
2. Program Comprehension Challenge = No (GP-N) – PR review comments that were unrelated to program comprehension challenge.

For each of the PR attributes, the null hypothesis and alternate hypothesis were:

H_0: The distributions of the PR attribute of the two groups (GP-Y and GP-N) are equal.

H_1: The distributions of the PR attribute of the two groups (GP-Y and GP-N) are not equal.

The statistical hypothesis test results are summarized in Table 5.2.

PR Attribute	Program Comprehension Challenge Yes	No	Total	X^2	p-value
Commit File Status				2.291	.514
added	13	61	74		
modified	57	263	320		
removed	1	1	2		
renamed	0	4	4		
Commenter Association				7.775	.1
COLLABORATOR	6	32	38		
CONTRIBUTOR	53	227	280		
MEMBER	58	309	367		
NONE	14	36	50		
OWNER	0	13	13		

(a) Chi-squared test for categorical PR attributes.

	Program Comprehension Challenge Yes (GP-Y)		No (GP-N)				
PR Attribute	Sample Size	Median	Sample Size	Median	U	Z	p-value
PR Commits Count	131	3	617	4	42742	1.053	.293
PR Additions	131	554	617	415	39295	-0.498	.619
PR Deletions	131	44	617	48	40642	0.102	.919
PR Changed Files	131	13	617	13	41323	0.405	.686
Commit File Additions	71	17	329	17	11731	0.057	.954
Commit File Deletions	71	2	329	2	11807	0.146	.884
Commit File Changes	71	30	329	27	11726	0.052	.959
PR Issue Comments Count	131	4	617	4	42118	0.761	.447
PR Review Comments Count	131	17	617	18	40247	-0.074	.941
PR Commits Count Prior to Comment	71	1	329	2	12615	1.082	.279

(b) Mann-Whitney U test for discrete PR attributes.

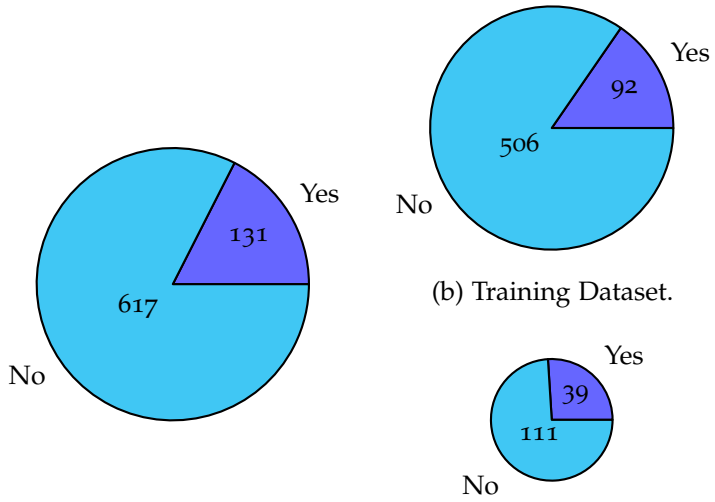
Table 5.2: Statistical hypothesis test results between pull request (PR) review comment attributes and program comprehension challenge labels. N.B. $\alpha = .05$.

5.2 DETECTING PROGRAM COMPREHENSION CHALLENGES WITH MACHINE LEARNING

The purpose of RQ2 was to evaluate whether ML would be an acceptable tooling mechanism:

> *RQ2. How accurately can a machine learning classifier detect program comprehension challenges in a pull request code review?*

This study evaluated ML classifiers built with three different ML algorithms, based on a target F-beta ($\beta = 0.5$) score of 0.6, a target precision rate of 70%, and a target recall rate of 60%. The breakdown of 748 samples of PR review comments labeled "*Yes*" for program comprehension challenge versus "*No*" for non-program comprehension challenge in the training and test datasets is summarized in Figure 5.2.



(a) Experiment Sample Dataset.

(b) Training Dataset.

(c) Test Dataset.

Figure 5.2: Training and test datasets sample size and split from experiment sample dataset.

5.2.1 *Machine Learning Classifiers Performance Results*

For each of the three algorithms, the effectiveness of different features and NLP tasks combinations were compared and cross-validated with the training dataset split in 5-fold for hyperparameters tuning, and then the performance was evaluated against the test dataset.

With the appropriate features and NLP tasks, both logistic regression and linear SVM with SGD learning achieved the target score. Moreover, all algorithms performed above the target for both labels when examined for precision. Nevertheless, a closer inspection of the table showed that only linear SVM with SGD learning was able to maintain an acceptable recall rate of more than 60% when detecting program comprehension challenges and outperformed the other two algorithms in most metrics apart from the precision rate for the label "*Yes*", and recall for the label "*No*."

When the results for different combinations of features were compared, it showed that with the features "body," "comment is by author," and "dialogue act classification" the ML classifiers performed better than with other combinations of features, except for the multinomial naive Bayes, which performed better with only "body" and "dialogue act classification." Furthermore, the feature weightings applied by the ML algorithm that produced the highest F-beta score were found for the three features "body" versus "comment is by author" versus "dialogue act classification." The weightings were four versus two versus one, respectively.

The NLP task combination involving the removal of stop words, lemmatization, and tf-idf consistently performed the best with the highest F-beta score across all three algorithms. The precision rates were also higher than other NLP tasks, apart from linear SVM with SGD learning with the combination of retaining stop words and tf-idf, which performed at a higher precision rate of 75%, but with a considerably lower recall rate of merely 23.1%.

The relevant performance results comparing the different ML classifiers trained with different combinations of ML algorithms, features, and NLP preprocessing on the PR review comments body text are exhibited in Table 5.3.

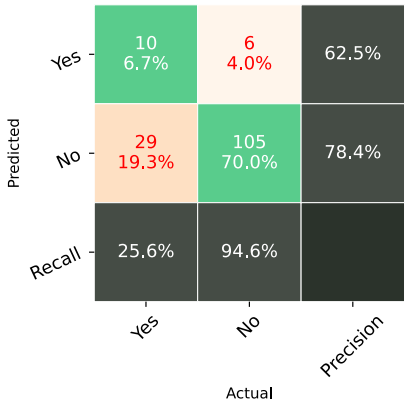
Table 5.3: Machine learning classifiers performance comparison for different combinations of learning algorithms, features and natural language processing (NLP) tasks. N.B. $\beta = 0.5$, Lowest → Highest. Glossary: support vector machine (SVM), stochastic gradient descent (SGD), and term frequency-inverse document frequency (tf-idf).

Learning Algorithm	**Program Comprehension Challenge Detection**				
Feature(s)	**Yes**			**No**	
NLP task(s) on Pull Request Review Comment Body Text	**F-Beta**	**Precision**	**Recall**	**Precision**	**Recall**
Multinomial Naive Bayes					
Body Text + Comment Is by Author + Dialogue Act Classification					
Bag-of-words	0.324	0.36	0.231	0.76	0.856
Stemming + tf-idf	0.421	0.529	0.231	0.774	0.928
Retain stop words + tf-idf	0.33	0.462	0.154	0.759	0.937
Remove stop words + Lemmatization + tf-idf	0.485	0.625	0.256	0.784	0.946
Body Text + Comment Is by Author					
Remove stop words + Lemmatization + tf-idf	0.467	0.588	0.256	0.782	0.937
Body Text + Dialogue Act Classification					
Remove stop words + Lemmatization + tf-idf	0.505	0.667	0.256	0.785	0.955
Logistic Regression					
Body Text + Comment Is by Author + Dialogue Act Classification					
Bag-of-words	0.0	0.0	0.0	0.74	1.0
Stemming + tf-idf	0.317	0.667	0.103	0.757	0.982
Retain stop words + tf-idf	0.526	0.714	0.256	0.787	0.964
Remove stop words + Lemmatization + tf-idf	0.683	0.76	0.487	0.84	0.946

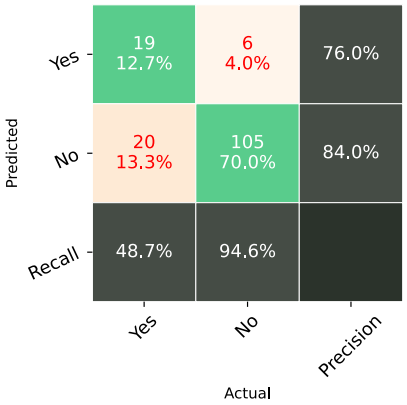
Table 5.3 continued from previous page

Learning Algorithm	**Program Comprehension Challenge Detection**				
Feature(s)	**Yes**			**No**	
NLP task(s) on Pull Request Review Comment Body Text	**F-Beta**	**Precision**	**Recall**	**Precision**	**Recall**
Body Text + Comment Is by Author					
Remove stop words + Lemmatization + tf-idf	0.647	0.72	0.462	0.832	0.937
Body Text + Dialogue Act Classification					
Remove stop words + Lemmatization + tf-idf	0.611	0.696	0.41	0.819	0.937
Linear SVM with SGD Learning					
Body Text + Comment Is by Author + Dialogue Act Classification					
Bag-of-words	0.409	0.433	0.333	0.783	0.847
Stemming + tf-idf	0.352	0.625	0.128	0.761	0.973
Retain stop words + tf-idf	0.517	0.75	0.231	0.783	0.973
Remove stop words + Lemmatization + tf-idf	0.726	0.743	0.667	0.887	0.919
Body Text + Comment Is by Author					
Remove stop words + Lemmatization + tf-idf	0.702	0.727	0.615	0.872	0.919
Body Text + Dialogue Act Classification					
Remove stop words + Lemmatization + tf-idf	0.659	0.688	0.564	0.856	0.91

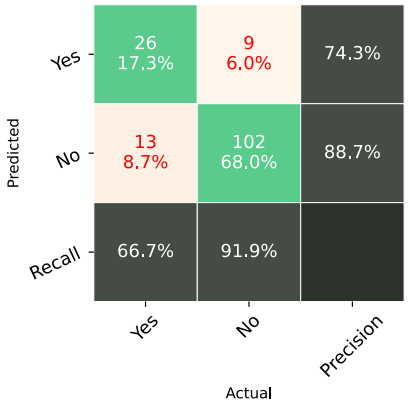
The confusion matrices for all three algorithms with the combination of the features "body," "comment is by author," and "dialogue act classification" and NLP tasks of removing stop words, lemmatization and tf-idf are presented in Figure 5.3.



(a) Multinomial naive Bayes.



(b) Logistic regression.



(c) Linear support vector machine with stochastic gradient descent learning.

Figure 5.3: Confusion matrices for the machine learning algorithms.

MULTINOMIAL NAIVE BAYES Multinomial naive Bayes failed to detect 29 instances of program comprehension challenges (false negatives), resulting in the lowest recall rate of 25.6% for "*Yes*." The low recall rate indicated that 74.4% of program comprehension challenges were not detected by the ML classifier with this algorithm. On the contrary, the recall rate for "*No*" was one of the highest at 94.6%.

LOGISTIC REGRESSION Logistic regression produced similar results, with a slightly better but still underperformed recall rate for "*Yes*" at 48.7%. Interestingly, the metrics indicated that this algorithm had the highest precision rate and detected the true positives with the highest certainty at 76%. However, this high precision rate was heavily impaired by the low recall rate, which indicated that 51.3% of the false negatives were not recognized by the ML classifier. In addition, the algorithm performed at the same level as multinomial naive Bayes for the recall rate for "*No*."

LINEAR SVM WITH SGD LEARNING Lastly, linear SVM with SGD learning detected program comprehension challenges with a precision rate of 74.3% and a recall rate of 66.7%. On average, this algorithm had the most balanced performance, and it scored over 70% for most metrics except for the recall rate for "*Yes*," ranked second in precision for "*Yes*," had over 90% recall rate for "*No*," and was at the top for all other metrics. As shown in the confusion matrix, this algorithm correctly detected 26 instances of program comprehension challenges from the test dataset, which was higher than multinomial naive Bayes (10 instances) and logistic regression (19 instances).

5.3 TYPES OF PROGRAM COMPREHENSION CHALLENGES DURING CODE REVIEWS

The program comprehension challenges were analyzed to answer RQ3:

> *RQ3. What types of program comprehension challenges do reviewers face?*

A content analysis was conducted on 384 program comprehension challenges, including the 131 instances already identified from the experiment sample dataset, to categorize the review comments that reflected the types of information needs encountered during code reviews. Table 5.4 provides a few key examples of how the raw review comments data were classified into the categories.

5.3.1 *Results*

Out of the 748 experiment sample dataset, 17.5% of the review comments were found to reveal program comprehension challenges. During the content analysis, the type "*design rationale for the implementation*" tended to be more frequently mentioned in

Category	Example
Intended program behavior	Do we need to clear the list here explicitly?
Developer's intention	Why did you do this?
Design rationale for the implementation	Why not use the client that's already created for the integration tests?
Developer familiar with the code	Is there a way to generally say this magnitude is "before" the other one if they have different systems?
Author of the code	Not available

Table 5.4: Key examples for each category.

the review comments, at a rate of more than 44%. In addition, "*developer's intention*" was found in more than 28% of the review comments, closely followed by "*intended program behavior*" at 25.3%. In contrast, "*developer familiar with the code*" occurred less than 2%, and no cases inquiring about "*author of the code*" were found. The proportions of the types of program comprehension challenges observed is shown in Figure 5.4.

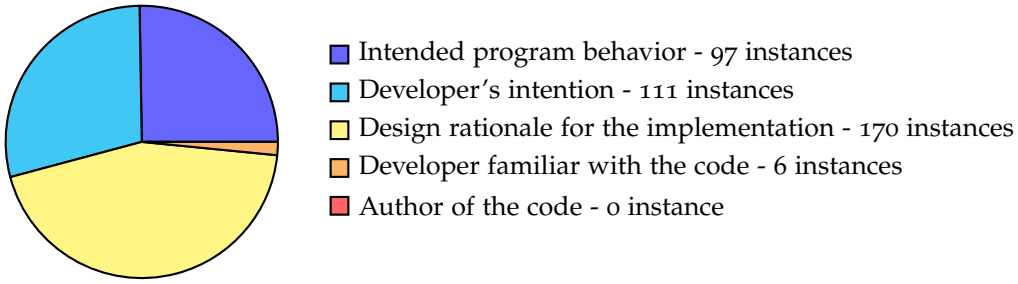


Figure 5.4: Frequencies of the types of program comprehension challenges encountered.

5.4 CAUSES OF PROGRAM COMPREHENSION CHALLENGES

The final goal was to investigate the possible impedances of program comprehension:

RQ4. Why do reviewers face program comprehension challenges?

A content analysis of two datasets was conducted to answer this RQ. One set contained 384 samples of program comprehension challenges that captured 94 distinct topics. The other sample size was the same, with 96 distinct topics included. These topics

were the condensed abstraction of the meaning units, based on the integrated comprehension model knowledge structures that include program domain knowledge, problem domain knowledge, programming plans, and rules of discourse [53].

5.4.1 *Results*

Chi-squared tests [85] found that five topics correlated positively with program comprehension challenges, and five topics correlated positively with non-program comprehension challenges. The remainder of the topics did not show any correlation with program comprehensibility. The detailed results for the topics showing significant correlation are presented in Table 5.5.

	Program Comprehension Challenge					$\alpha = .05$
Topic	**Yes**	**No**	***df***	***N***	X^2	***p*-value**
Program Logic	77	39	1	748	13.556	< .05
Code Design	45	25	1	748	6.001	< .05
Defensive Coding	29	13	1	748	6.272	< .05
Condition Checking	15	1	1	748	12.38	< .05
Concurrency	13	2	1	748	8.147	< .05
Code Readability	0	15	1	748	15.299	< .05
Coding Convention	5	18	1	748	7.461	< .05
Naming Intention	0	10	1	748	10.132	< .05
Code Duplication	5	15	1	748	5.067	< .05
Performance	3	12	1	748	5.454	< .05

Table 5.5: Topics and program comprehensibility with significant correlation.

Table 5.6 provides a few key examples that demonstrate how the raw review comments data were condensed into the topics with significant correlation.

Topic	Program Comprehension Challenge	Example
Program Logic	Yes	Do we need to explicitly clear the list here?
Code Design	Yes	Why a 'Service'? Why an abstract class instead of an interface?
Defensive Coding	Yes	What happens if enum.toInt(value) returns null?
Condition Checking	Yes	Won't this always be true because in the previous line you are adding 'CGEO_PREFIX.length()'?
Concurrency	Yes	Can you expand on where the lock contention could occur? Do you mean scenario when Engine does not reach it's capacity limit?
Coding Readability	No	I think we should use new line for each field (I think it increases readability)
Coding Convention	No	For json we should always use underscore
Naming Intention	No	IMO 'getComponent(component)' reads a bit confusingly. Could we change the method name to something like 'getUIPart'?
Code Duplication	No	We already have getRandomShort() in Utils.
Performance	No	Creating a new transformer for every message looks very inefficient.

Table 5.6: Key examples for each topic with significant correlation.

6

DISCUSSION

This chapter delves into the key findings for each RQ, compares the results the findings in the literature review, notes the implications of the findings, and provides recommendations for future work.

6.1 PULL REQUEST ATTRIBUTES AFFECTING PROGRAM COMPREHENSIBILITY

The results for RQ1 – *What are the attributes of pull requests that affect program comprehensibility?* – indicate that, during code reviews, the set of identified PR attributes a PR author is in control of and the code metrics related to LOC, including the number of lines added, deleted or modified, have no effect on program comprehensibility. In comparison, only the dialogue act of the PR review comment and whether the comment was by the PR author correlated with the instances of program comprehension challenges.

Prior studies have noted the importance of LOC for estimating the effort required for program comprehension [19, 60], but, no data was found on the association between code metrics and program comprehensibility during code reviews in the review of the literature. Since code reviews primarily focus on and comprehend code changes, the additions, deletions, and modifications to the number of lines are related measurements of LOC in the context of code reviews. The following sections discuss LOC and the subset of the attributes measuring the number of code changes made.

NUMBER OF LINES OF CODE CHANGED One part of RQ1 set out the aim of assessing the importance of LOC in a PR:

> *RQ1a. How does the number of lines of code changed in a pull request affect program comprehensibility?*

Herraiz and Hassan [60] claimed that LOC is a useful metric to estimate the comprehension effort. However, the difference observed in distributions of the number of lines changed between GP-Y and GP-N in this study was not significant and no evidence supporting the claim was detected. Instead, the results are in line with

the findings from Ajami *et al.* [61], who claimed that LOC may not be an important confounding cause for comprehension efforts.

The statistical hypothesis test results of this study contribute a clearer understanding of program comprehension challenges in a code review that may not be affected by the number of lines changed. The reason for this is not apparent, but it might be because LOC does not weight each line equally [61, 62]. Another possible explanation for this is that, while the comprehension effort may increase during a review of the PR, since the data gathered only reflects instances of comprehension challenges and lacks information regarding the amount of time a reviewer spent on a PR, the results cannot confirm the overall program comprehension effort.

NUMBER OF COMMITS The other part of RQ1 was set to identify the effect of the number of commits on program comprehension:

> *RQ1b. How does the number of commits in a pull request affect program comprehensibility?*

The results of this study do not show any significant increase in the distributions of the number of commits between the PR review comments that reveal program comprehension challenges and non-program comprehension challenges.

Furthermore, this implication further supports the idea that program comprehension challenges are not affected by the amount of code changed. It may be that the code reviewers benefit from the code review user interface (UI) condense changes across all the commits to date when the reviewer reviews the PR.

DIALOGUE ACT CLASSIFICATION The results from this feature selection build on existing evidence that software engineers often use interrogative words to express information needs when facing program comprehension challenges. Letovsky [68] suggested four fundamental interrogative words for comprehension, namely "why," "how," "what," and "whether."

These interrogative words can be identified with the application of an ML classifier to classify the dialogue act of PR review comments [73]. Furthermore, in the review of the literature, it was found that the types of program challenges are categorized in the form of "*whQuestion*" [9, 14]. A comparison of the findings confirms that dialogue act "*whQuestion*" and "*ynQuestion*" have a significant higher usage than expected for the instances of program comprehension challenges.

COMMENT IS BY AUTHOR One unanticipated finding was that, from the 131 samples, there were 10 instances of program comprehension challenges encountered by the PR authors themselves. An inspection of those instances revealed that this finding may be explained by the fact that there are scenarios where an author may be responding to suggested code changes in the review feedback, and a program comprehension challenge arises from an understanding of the broader codebase beyond the scope of the code changes in the PR, or the challenge may relate to an understanding of the intention or side effect of the suggested changes. This finding contributes a clearer understanding of the implication that, during the code review, the program comprehension activity is a joint exercise for both the reviewer and the author.

INSIGNIFICANCE OF COMMIT FILE STATUS This study did not detect any evidence for a correlation between the type of commit file status and program comprehension. There are several possible explanations why file addition, deletion, renaming, or modification do not affect program comprehensibility during a PR code review.

One possible explanation might be the fact that the organization of files in a code repository is often conventional, particularly in some programming languages such as Java, so that a file contains only a class or an interface, and it is placed in the folder structure according to its namespace.

Another possible explanation is that a code review UI, regardless of whether it is the web interface on GitHub or in an IDE, highlights the code changes but deemphasizes the file structure. The code changes include line additions, deletions, and modifications to parts of a line. The file path or file structure in the code review interface is often folded and collapsed. These highlights attract the focus of code reviewers and are the primary artifacts for program comprehension (as illustrated in Figure 2.2).

INSIGNIFICANCE OF OTHER ATTRIBUTES BEYOND CONTROL BY AN AUTHOR Surprisingly, program comprehension challenges also did not show any significant relationship with a higher number of PR issue comments or review comments. This somewhat contradictory result may be due to a possible scenario where a PR contains too many comments. Since the discussion may have been sidetracked, became too complicated, or required excessive cognitive load, a reviewer might avoid becoming

involved in the discussion or approve the PR without understanding the code changes, especially if the PR author is recognized as a senior contributor.

6.2 DETECTING PROGRAM COMPREHENSION CHALLENGES WITH MACHINE LEARNING

No studies that used ML to classify PR review comments were found in the literature review, although some studies have classified other types of textual content with ML and NLP with satisfactory accuracy [46, 47].

RQ2 – *How accurately can a machine learning classifier detect program comprehension challenges in a pull request code review?* – was set to assess the accuracy of ML in detecting program comprehension challenges in PR code review comments. The results of the assessment of the ML performance indicate that, with the described features, NLP preprocessing, and ML algorithm to train an ML classifier, the classifier achieved a precision rate of 74.3% and a recall rate of 66.7% in detecting program comprehension challenges. Therefore, it can be assumed that the program comprehension challenges can be accurately detected with ML.

SVM PERFORMANCE FOR TEXT CATEGORIZATION Among the three ML algorithms evaluated, linear SVM with SGD learning consistently outperformed the other algorithms. These results reflect the findings of Zhang and Oles [90] and Joachims [92], who also found that SVM performed better than two other algorithms.

LACK OF CONTEXT FROM A SINGLE REVIEW COMMENT POST One of the issues that emerged from these findings is that a single review comment post, taken out of context of a discussion, may not be sufficient for a human to determine whether it expressed program comprehension challenges, for example: "Can we use isNotBlank()?" Five instances were noted and discussed during the manual labeling reconciliation meetings, and all five were manually labeled as "*No*." Therefore, the results of the training and test datasets have to be interpreted with caution, as some legitimate instances of comprehension challenges might have been considered irrelevant. Consequently, the ML classifiers may be somewhat limited by the information provided in review comments, and a shorter review comment without sufficient context may not be detected as accurately as a longer review comment.

EFFECTIVENESS OF NLP The preliminary finding suggests that a classifier's performance is significantly impacted by the feature extraction and selection in NLP tasks on the body text, and other PR attributes did not seem to affect the performance to the same extent. This finding also helps to understand that program comprehension challenges expressed by software engineers may be different from a regular review comment. The ML was successful since it was able to detect program comprehension challenges at an acceptable precision for the content analysis.

6.3 TYPES OF PROGRAM COMPREHENSION CHALLENGES DURING CODE REVIEWS

Relating to RQ3 – *What types of program comprehension challenges do reviewers face?* – studies by Maalej *et al.* [9] and Ko *et al.* [14] both evaluated problems encountered while attempting to understand code; however, no studies focusing on code review activity have been described. This subsection discusses the similarities and differences in the findings in this study as compared to previous studies.

DESIGN RATIONALE FOR THE IMPLEMENTATION In contrast to earlier findings, this study found that "*design rationale for the implementation*" occurred more frequently, significantly more than the other types. In the scenarios where this type of question was asked, the reviewer comments often revealed that the reviewer understood the motive and the effect of the code change; however, they did not clearly understand the rationale behind the implementation and asked the author for clarification, for example: "Any reason why to use 'grep' over 'findAll'?"

This finding is similar to what was observed in studies done by Dabbish *et al.* [97], and provides some support for the conceptual premise that PR code review is a useful tactic for managing incoming contributions to ensure code quality.

DEVELOPER FAMILIAR WITH THE CODE The results might suggest that there was less need to discover who has had experience with a specific code. However, based on the findings of similar studies, a more plausible explanation is that the design of the PR code review UI on GitHub reduces the effort to navigate this information.

Maalej *et al.* [9] reported that almost 90% of the developers surveyed encountered this problem at least once a month when attempting to understand the code of others, and 48.5% of them encountered it more frequently, on a weekly or daily basis. This

frequency differs from the findings in this study, with less than 2% of the review comments categorized as this type of challenge.

The information to view all the contributors of a file in a PR on the GitHub web UI, is only two clicks away, and one of the fundamental design principles for modern version control systems is traceability to track contributors who worked on the piece of code.

Although Maalej *et al.* [9] reported that dedicated program comprehension tools are rarely used, this study found important implications for the direction of developing program comprehension tools. This raises intriguing questions regarding the nature and extent of those types of tools, such as whether they would be more effective and convenient to software engineers if embedded in the software development workflow related to the information they provide.

INTENDED PROGRAM BEHAVIOR Maalej *et al.* [9] and Ko *et al.* [14] both observed that "*intended program behavior*" is the type of information software engineers sought most frequently; 85% of the developers reported encountering this problem weekly and, and approximately 50% of them, even daily. On the contrary, the levels observed in this study are far below those observed in previous studies.

This inconsistency may be due to the top-down knowledge of programming plans regarding what the code changes are supposed to do that may have been primarily explained in the PR description, or by the issue referenced in the PR, resulting in fewer questions regarding the application domain. This finding also suggests that it is possible, therefore, that the bottom-up comprehension model is less frequently used during a code review in a PR.

AUTHOR OF THE CODE Interestingly, in all 384 samples in this study, no single instance related to this type of information need was found, which is considerably lower than the level reported by Maalej *et al.* [9], who found that 38.5% of developers encountered this issue often, sometimes as much as once a week.

Although a PR may involve multiple contributors, there can only be one PR author. These results are likely to be related to clear ownership and accountability, which significantly reduces the likelihood of this type of information need during code reviews.

6.4 CAUSES OF PROGRAM COMPREHENSION CHALLENGES

Relating to RQ4 – *Why do reviewers face program comprehension challenges?* – "*program logic*," "*code design*," "*defensive coding*," "*condition checking*," and "*concurrency*" were found to cause program comprehension challenges during code reviews. These are all related to the bottom-up comprehension approach and the program domain knowledge described in the integrated comprehension model by von Mayrhauser and Vans [53]. "*Program logic*," "*code design*," "*defensive coding*," and "*condition checking*" are subsets of the control sequence, and "*concurrency*" is a subset of data flow.

In comparison, review comments unrelated to any program comprehension challenges mostly discussed topics related to the top-down model knowledge. "*Code readability*," "*coding convention*," and "*naming intention*" are part of rules of discourse, and "*code duplication*" is part of implementation plans. "*Performance*" was the only topic related to the program domain knowledge in the program model.

It is difficult to explain this result, but it might be related to code review being a static analysis. The code review UI does not support navigating through call stacks or checking the use of related code changes. Runtime information is also not readily available to the reviewers. Maalej *et al.* [9] reported the importance of that information in assisting with program comprehension. Hence, a simple code change might become difficult to comprehend without the information to illustrate how the code changes fit the higher-level programming plan. For instance, "Do we need to explicitly clear the list here?", commented on code change `allIpList.clear()`; while the code change is straightforward, but the way this change impacts the wider programming plan is unclear.

In the discussion in a previous section it was mentioned that the bottom-up comprehension model might be less frequently applied. A study by Shaft and Vessey [51] also revealed that software engineers tend to avoid the bottom-up approach to reduce cognitive load. This combination of findings in this study and in the study by Shaft and Vessey [51] supports the conceptual premise that the bottom-up comprehension model approach is a last resort when the reviewers focus on program domain knowledge. Furthermore, these scenarios signify missing supplementary information on the programming plans from the top-down model or problem domain knowledge from the situation model; thus, the reviewers are more likely to encounter program comprehension challenges.

6.5 LIMITATIONS

One internal threat and one external threat may potentially influence this study's findings and two limitations could affect reliability.

6.5.1 *Threat to Internal Validity*

Instrumentation design is a known threat to internal validity. The content analysis dataset of program comprehension challenges contained 131 manually labeled samples and 253 samples labeled by the trained ML classifier. Notwithstanding the differences in the labeling, this threat is mitigated by the acceptable precision (74.3%) and recall (66.7%) rates of the ML classifier.

6.5.2 *Threat to External Validity*

Sampling bias is a known threat to external validity. It is impossible to assess all code repositories of different programming languages; therefore, it is not known whether different programming languages pose different comprehension challenges since they have different characteristics, such as code file structure convention or syntax readability. Although the generalizability of the results could be limited by some hidden factors, to counter the threat, the experiment dataset purposive sampled PR review comments from repositories containing predominantly Java code, as it has been one of the top three popular languages for the past six years [34].

6.5.3 *Small Number of Samples*

Compared to the number of PR review comments available on GitHub, the experiment sample dataset was limited in the number of PR review comments that this study could investigate. This small sample size can limit the generalizability of the results. However, this limitation is mitigated by simple random sampling from a large experiment dataset of more than 1,036,743 review comments, which diversified the samples in the pool.

Moreover, out of the 748 samples, commit file information could be retrieved from only 400 samples. The missing information was caused by the scenarios where there

was a subsequent commit that modified the line referenced by the review comment; the GitHub API removes the original commit from the PR commit list, resulting in missing commit file information in the data collection. The smaller sample size of only 400 samples for the commit file information lowered the confidence level to 95% with a confidence interval of 4.9, compared to the other attributes that provided a confidence level of 99% with a confidence interval of 4.72.

6.5.4 *Researcher Bias*

During the manual labeling procedures, the interpretation of comprehension challenges is subjective, so a researcher's preconceptions, such as software development knowledge, experience, or programming skills, can induce bias [96].

The threat to the data reliability of the labeling of the experiment sample dataset used for ML training was mitigated by measuring the intercoder reliability coefficients [83] of the labeled data through three independent coders.

For the content analysis datasets, this limitation was dealt with in three aspects. One aspect was by limiting the filtering of the raw data to only repositories using Java, given its popularity and the researcher's programming skill, so that the code changes in the PR could be understood. Another aspect was a similar method that filtered out non-English review comments with the language detector library cld2-cffi [78]. The last aspect was being aware of the preconceptions during the content analysis, and, in addition to using intuition for generating a topic, iteratively refining the topics and investigating unfamiliar key terms that appeared in the review comments.

6.6 SUMMARY

The most prominent finding to emerge from this study is that, based on a performance comparison of three ML algorithms implemented to evaluate the precision and recall rates in detecting program comprehension challenges, ML can achieve at least 74.3% precision and 66.7% recall. The results suggest that the accuracy can be achieved through the combination of linear SVM with SGD learning and NLP tasks of dialogue act classification, stop words removal, lemmatization, and tf-idf.

By analyzing the instances of review comments that indicated failure to understand the code changes, this study confirmed that LOC and the number of commits made

no significant difference to program comprehensibility. Also, interrogative words are often used, and the results showed the communication often expressed in dialogue act "*whQuestion*" and "*ynQuestion*."

One of the more significant findings to emerge from this study is that the types of information needed for understanding codebase when editing and extending source code during software maintenance differ noticeably from what is required to understand code changes in a code review.

However, determining the programming plan to articulate the intended program behavior is the main problem encountered during software maintenance, but it ranked only third in code reviews and accounted for only 25.3% of the challenges.

Furthermore, the content analysis revealed that, in general, missing supplementary top-down knowledge regarding the programming plans and situation model knowledge regarding the problem domain behavior are common causes that impede reviewers' comprehension in code changes.

6.6.1 *Implications*

There are four implications for future research from this study:

1. *Discussion thread is an important context to improve the detection rate of the ML classifier for program comprehension challenges.*

   This finding has important implications for the development of ML classifiers with the discussion threads from PR reviews to provide the background context of the conversation. In future investigations, it might be possible to use the GitHub GraphQL API, which offers an object PullRequestReviewThread that provides all comments in a PR, organized in a threaded list. Further studies that consider these data sources will have to be conducted.

2. *LOC does not affect program comprehensibility during code reviews.*

   The results in this study may support the hypothesis that LOC is not a critical confounding cause for program comprehensibility during code reviews [61]. To develop a full picture of the correlation of program comprehensibility and other code metrics, such as HCM and CCM, additional studies will be required to obtain the information on the code changes from GitHub's commits API and a snapshot of the code repository. This additional information will help to address many

unanswered questions regarding the characteristics of the code changes, such as the number of operators and operands or the number of nodes or connected components.

3. *Code review requires a different set of program comprehension information needs to what is required for software maintenance.*

   This study raises the possibility that the information needs for understanding codebase when editing and extending the source code during software maintenance [9, 14] differs noticeably from understanding code changes in a code review. Clear accountability and ownership of PR almost eradicate the need to discover who wrote the piece of code. In addition, the UI in PR code review significantly reduces the effort to discover who has experience with the code, as that information is easily accessible; evidently, less than 2% of the review comments sought this information. Reviewers often wonder why a piece of code is implemented in a certain way, and this information need is seldom encountered when writing code. Furthermore, to better understand the implications of these results, future studies could investigate a larger sample of program comprehension challenges and develop an ML that automatically categorizes the types of program comprehension challenges.

4. *Distinguishing soft suggestions from program comprehension challenges is essential.*

   One of the issues that emerged from manual labeling is that, when changing code, many reviewers are polite and indirect with their communication of suggestions with concrete implementation alternatives, for example: "add a message as the second argument?" This observation may be explained by the fact that the open-source community and the full transparency in the code review causes the reviewer to communicate with caution and reduce the possibility of potential bias or prejudice. It is encouraging to compare this observation with a study by Dabbish *et al.* [97], who found that the transparent feedback on GitHub drives the desire of reviewers to manage their reputation and status in the online community. Nine instances of soft suggestions were noted and discussed during the reconciliation meetings to establish the manual labeling guideline. Those soft suggestions were treated as irrelevant to program comprehension challenges, as the reviewer displayed confidence in understanding the code changes.

6.6.2 *Guidelines*

There are three guidelines from this study for practice and future research:

1. *Review comments categorization should consider the linear SVM with the SGD learning algorithm.*

   The ML algorithm consistently achieved acceptable precision and recall rates from the ML classifier performance results compared to two other algorithms evaluated. Further research on review comment categorization focusing on this ML algorithm is suggested.

2. *Feature engineering on English review comments should consider NLP tasks including dialogue act classification, stop words removal, lemmatization, and tf-idf.*

   The combination of NLP tasks that extracted and filtered the textual features from review comments consistently outperformed other combinations. Future research in this field should utilize the NLP task combination as a baseline for feature engineering.

3. *Integrate top-down knowledge into the workflow.*

   Based on the findings for the types of program comprehension challenges, the information need for "*author of the code*" and "*developer familiar with the code*" are significantly lower during code reviews, as this information is the background context and part of the PR workflow. The findings for the causes of program comprehension challenges suggest that the reviewers seek top-down knowledge that fit with program domain knowledge. Practitioners should consider developing future program comprehension tools embedded as part of the workflow, to allow software engineers to access the relevant top-down knowledge of programming plans only a few clicks away from their current task.

7

CONCLUSION

This study aimed to identify the accuracy of detecting program comprehension challenges faced by software engineers when reviewing code changes in PRs with the application of ML, and, furthermore, to analyze the specific challenges surrounding program comprehension to compare to the challenges of writing and developing code during software maintenance, and to discover the common causes of the challenges.

7.1 REMARKS AND OBSERVATIONS

This research clearly illustrates that ML can accurately detect program comprehension challenges, but it also raises the matter of the importance of knowing the context of a PR review discussion thread.

The sampling focused predominantly on Java code, and, while this bias limits the generalizability of the results, these experiments confirmed that LOC and the number of commits had no significant impact on program comprehensibility during code reviews. Moreover, this study has identified the types of program comprehension challenges specific to code review activity. Furthermore, the relevance of programming plans and the top-down knowledge required during code review is clearly supported by the current findings.

To address the limitations discussed in Section 6.5, further research should incorporate more sample data, collect discussion threads to provide more context for a PR review comment, and expand the scope to different programming languages. Alternatively, instead of individual PR review comments, an entire PR with all review threads as a meaning unit or the actual code changes associated with the code review could be analyzed.

This study has provided an in-depth insight into program comprehension and the comprehension strategies software engineers currently apply during code reviews. Three ML algorithms and four combinations of NLP tasks were evaluated, and recommendations for future research in the area of PR review comments were provided. The proven concept for the ML classifier developed in this study may assist in the

detection of comprehension issues and provide analytics to software development teams, assisting them to prioritize areas of improvement.

The ML classifier can be further optimized. The NLP tasks and hyperparameter tuning explored relatively simple combinations, and more sophisticated NLP techniques or different ranges of hyperparameters can be further explored. Further research can even experiment with different machine learning architecture, such as neural networks. In addition, the ML classifier can be improved for the classification of the types of program comprehension challenges to better understand the implications of the results. Future studies could investigate a larger sample of instances of program comprehension challenges and develop an ML classifier that automatically categorizes the types of program comprehension challenges.

These contributions enable a decrease in cognitive load for program comprehension, allow software engineers to focus on writing new code and developing new features, and more efficient code review and pull-based development. Furthermore, the common causes of comprehension challenges investigated in this study serve as a basis for suggestions to guide software engineers on the information they provide when creating PRs, rendering code review a more enjoyable and productive exercise. Consequently, time spent on trivial impediments that affect program comprehensibility can be minimized and development velocity can be maximized, hence, reducing the overall cost of software development.

A

DEPENDENT PYTHON PACKAGES

Python Package	Dependent Version	Primary Purpose	Package URL
cld2-cffi	0.1	Language detection to select only English review comments.	https://pypi.org/project/cld2-cffi/
matplotlib	3.3	For creating ML classifier performance visualizations.	https://pypi.org/project/matplotlib/
nltk	3.5	For creating a Dialogue Act classifier.	https://pypi.org/project/nltk/
pandas	1.1	For processing the *.csv datasets.	https://pypi.org/project/pandas/
pymongo	3.11	Data retrieval from GHTorrent's MongoDB.	https://pypi.org/project/pymongo/
scikit-learn	0.23	ML library with a comprehensive set of algorithms.	https://pypi.org/project/scikit-learn/
seaborn	0.11	For creating ML classifier confusion matrix visualizations.	https://pypi.org/project/seaborn/
sshtunnel	0.1	Integration with GHTorrent's MongoDB via an SSH tunnel.	https://pypi.org/project/sshtunnel/
tqdm	4	Progress bar visualization for the data collection process.	https://pypi.org/project/tqdm/

Table A.1: List of dependent Python packages.

B

GHTORRENT BIGQUERY QUERY

The GHTorrent BigQuery dataset was produced in April 2018, with a cutoff date of 2018-03-31 for all data previously collected. When querying data from it, the term recent refers to activities within in the past three months (i.e., from 2018-01-01 to 2018-03-31), medium-term means activities from 2016-01-01 to 2017-12-31.

Listing B.1: GHTorrent BigQuery repository selection criteria.

```
SELECT
  p.id AS project_id,
  p.url AS project_url,
  p.description,
  pc.latest_commit_date,
  pc.mdm_term_commit_cnt,
  pc.mdm_term_distinct_author_cnt,
  pc.mdm_term_distinct_committer_cnt,
  pc.recent_commit_cnt,
  pc.recent_distinct_author_cnt,
  pc.recent_distinct_committer_cnt,
  prstats.latest_pull_request_history_date,
  prstats.mdm_term_pull_request_cnt,
  prstats.recent_pull_request_cnt,
  p.LANGUAGE AS project_language,
  pl.language AS project_language_details_language,
  pl.bytes AS project_language_bytes,
  pl.created_at AS project_language_created_at,
  p.forked_from,
  pr.id AS pull_request_id,
  pr.pullreq_id,
  pr.intra_branch,
  prc.user_id,
  prc.comment_id,
  prc.position,
  prc.body,
  prc.commit_id,
  prc.created_at
FROM `ghtorrent-bq.ght_2018_04_01.projects` AS p
```

```
INNER JOIN (
  -- Projects that are active and sustaining.
  SELECT
    pc.project_id,
    MAX(c.created_at) AS latest_commit_date,
    COUNT(DISTINCT author_id) AS mdm_term_distinct_author_cnt,
    COUNT(DISTINCT committer_id) AS mdm_term_distinct_committer_cnt,
    COUNT(commit_id) AS mdm_term_commit_cnt,
    COUNT(DISTINCT CASE WHEN c.created_at >= '2018-01-01' THEN author_id END) AS
↪ recent_distinct_author_cnt,
    COUNT(DISTINCT CASE WHEN c.created_at >= '2018-01-01' THEN committer_id END) AS
↪ recent_distinct_committer_cnt,
    COUNT(CASE WHEN c.created_at >= '2018-01-01' THEN commit_id END) AS
↪ recent_commit_cnt
  FROM `ghtorrent-bq.ght_2018_04_01.project_commits` AS pc
    INNER JOIN `ghtorrent-bq.ght_2018_04_01.commits` AS c ON c.id = pc.commit_id
  WHERE c.created_at >= '2016-01-01' -- Medium Term activity
  GROUP BY pc.project_id
) AS pc ON pc.project_id = p.id
INNER JOIN (
  -- Uses Pull Request, and have recent and medium term activities.
  SELECT
    base_repo_id,
    MAX(prh.created_at) AS latest_pull_request_history_date,
    COUNT(DISTINCT pr.id) AS mdm_term_pull_request_cnt,
    COUNT(DISTINCT CASE WHEN prh.created_at >= '2018-01-01' THEN pr.id END) AS
↪ recent_pull_request_cnt
  FROM `ghtorrent-bq.ght_2018_04_01.pull_requests` AS pr
    INNER JOIN `ghtorrent-bq.ght_2018_04_01.pull_request_history` AS prh ON
↪ prh.pull_request_id = pr.id
  WHERE prh.created_at >= '2016-01-01' -- Medium Term activity
  GROUP BY base_repo_id
) AS prstats ON prstats.base_repo_id = p.id
LEFT JOIN (
  -- Subquery to get projects that have Java as one of the prominent languages
  SELECT
    pl.project_id,
    pl.language,
    pl.created_at,
    pl.bytes
  FROM (
```

```
      SELECT
        pl.project_id,
        pl.created_at,
        SUM(bytes) AS total_bytes
      FROM (
        SELECT
          project_id,
          MAX(created_at) AS latest_refresh_date
        FROM `ghtorrent-bq.ght_2018_04_01.project_languages`
        GROUP BY project_id
      ) AS pl_latest
        INNER JOIN `ghtorrent-bq.ght_2018_04_01.project_languages` AS pl ON
  ↪ pl.project_id = pl_latest.project_id AND pl.created_at =
  ↪ pl_latest.latest_refresh_date
      GROUP BY pl.project_id, pl.created_at
    ) AS pl_latest_total_bytes
      INNER JOIN `ghtorrent-bq.ght_2018_04_01.project_languages` AS pl ON
  ↪ pl.project_id = pl_latest_total_bytes.project_id AND pl.created_at =
  ↪ pl_latest_total_bytes.created_at
    WHERE LOWER(pl.language) = 'java'
    AND pl_latest_total_bytes.total_bytes > 0
    AND pl.bytes / pl_latest_total_bytes.total_bytes > 0.5 -- Java is prominent.
  ) AS pl ON pl.project_id = p.id
  LEFT JOIN `ghtorrent-bq.ght_2018_04_01.pull_requests` AS pr ON pr.base_repo_id = p.id
  LEFT JOIN `ghtorrent-bq.ght_2018_04_01.pull_request_comments` AS prc ON
  ↪ prc.pull_request_id = pr.id
WHERE p.deleted = FALSE
AND (LOWER(p.LANGUAGE) = 'java' OR LOWER(pl.LANGUAGE) = 'java') -- Java is prominent.
AND (pc.mdm_term_commit_cnt - pc.recent_commit_cnt) >= 5
AND (
  (pc.mdm_term_distinct_author_cnt - pc.recent_distinct_author_cnt) >= 3
  OR (pc.mdm_term_distinct_committer_cnt - pc.recent_distinct_committer_cnt) >= 3
) -- At least 3 collaborators.
AND pc.recent_commit_cnt >=5
AND (pc.recent_distinct_author_cnt >= 3 OR pc.recent_distinct_committer_cnt >= 3)
AND (prstats.mdm_term_pull_request_cnt - prstats.recent_pull_request_cnt) >= 5
AND prstats.recent_pull_request_cnt >= 5
AND LOWER(p.description) NOT LIKE '%mirror of %'
AND prc.comment_id IS NOT NULL -- Need to have records with comments for the analysis.
ORDER BY pc.latest_commit_date DESC
```

REFERENCES

[1] A. von Mayrhauser, A. M. Vans, and A. E. Howe, "Program understanding behaviour during enhancement of large-scale software," *Journal of Software Maintenance: Research and Practice*, vol. 9, no. 5, pp. 299–327, 1997. DOI: https://doi.org/10.1002/(SICI)1096-908X(199709/10)9:5<299::AID-SMR157>3.0.CO;2-S.

[2] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *CASCON First Decade High Impact Papers*, ser. CASCON '10, Toronto, Ontario, Canada: IBM Corp., 2010, pp. 174–188. DOI: https://doi.org/10.1145/1925805.1925815.

[3] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, Sep. 2009. DOI: https://doi.org/10.1109/TSE.2009.28.

[4] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989. DOI: https://doi.org/10.1147/sj.282.0294.

[5] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00, Limerick, Ireland: ACM, 2000, pp. 73–87. DOI: https://doi.org/10.1145/336512.336534.

[6] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer - an investigation of how developers spend their time," in *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015, pp. 25–35. DOI: https://doi.org/10.1109/ICPC.2015.12.

[7] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, December 2006. DOI: https://doi.org/10.1109/TSE.2006.116.

[8] R. K. Fjeldstad and W. T. Hamlen, "Application program maintenance study: Report to our respondents," in *Tutorial on Software Maintenance*, G. Parikh and N. Zvegintzov, Eds. IEEE Computer Society Press, 1983, pp. 13–30.

[9] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Trans.Softw.Eng.Methodol.*, vol. 23, no. 4, 31:37, Sep. 2014. DOI: https://doi.org/10.1145/2622669.

[10] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *[1993] Proceedings Working Conference on Reverse Engineering*, Baltimore, MD, USA: IEEE, 1993, pp. 27–43. DOI: https://doi.org/10.1109/WCRE.1993.287781.

[11] M. Sulír, "Program comprehension: A short literature review," in *SCYR 2015: 15th Scientific Conference of Young Researchers*, 2015.

[12] M. L. Nelson, "A survey of reverse engineering and program comprehension," *ArXiv*, vol. abs/cs/0503068, 2005.

[13] P. Marvin Zelkowitz, M. Zelkowitz, A. Shaw, and J. Gannon, *Principles of Software Engineering and Design*, ser. Prentice Hall International Series in Computer Science. Prentice-Hall, 1979. [Online]. Available: https://books.google.co.nz/books?id=ctcmAAAAMAAJ (retrieved November 23, 2020).

[14] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *29th International Conference on Software Engineering (ICSE'07)*, May 2007, pp. 344–353. DOI: https://doi.org/10.1109/ICSE.2007.45.

[15] R. Tiarks, "What maintenance programmers really do: An observational study," in *Workshop on Software Reengineering*, Citeseer, 2011, pp. 36–37.

[16] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, Jul. 2006. DOI: https://doi.org/10.1109/MS.2006.105.

[17] M. M. Lehman, "Laws of software evolution revisited," in *Software Process Technology*, C. Montangero, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 108–124.

[18] J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*, ser. Fraunhofer IESE series on software and systems engineering. Springer, 2016. [Online]. Available: https://link-springer-com.ezproxy.auckland.ac.nz/book/10.1007%2F978-3-319-34177-4 (retrieved November 23, 2020).

[19] Y. Gil and G. Lalouche, "On the correlation between size and metric validity," *Empirical Software Engineering*, vol. 22, no. 5, pp. 2585–2611, 2017. DOI: https://doi.org/10.1007/s10664-017-9513-5.

[20] B. W. Boehm, "Software engineering economics," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 1, pp. 4–21, January 1984. DOI: https://doi.org/10.1109/TSE.1984.5010193.

[21] J. T. Nosek and P. Palvia, "Software maintenance management: Changes in the last decade," *Journal of Software Maintenance: Research and Practice*, vol. 2, no. 3, pp. 157–174, 1990. DOI: https://doi.org/10.1002/smr.4360020303.

[22] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT Professional*, vol. 2, no. 3, pp. 17–23, May 2000. DOI: https://doi.org/10.1109/6294.846201.

[23] M. Skelton and M. Pais, *Team Topologies: Organizing Business and Technology Teams for Fast Flow*. IT Revolution Press, 2019. [Online]. Available: https://books.google.co.nz/books?id=Pj-IDwAAQBAJ (retrieved November 23, 2020).

[24] J. Krüger, J. Wiemann, W. Fenske, G. Saake, and T. Leich, "Do you remember this source code?" In *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, Gothenburg, Sweden: ACM, 2018, pp. 764–775. DOI: https://doi.org/10.1145/3180155.3180215.

[25] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, Shanghai, China: ACM, 2006, pp. 492–501. DOI: https://doi.org/10.1145/1134285.1134355.

[26] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" In *2012 34th International Conference on Software Engineering (ICSE)*, Jun. 2012, pp. 255–265. DOI: https://doi.org/10.1109/ICSE.2012.6227188.

[27] Miryung Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in OOPL," in *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04.*, August 2004, pp. 83–92. DOI: https://doi.org/10.1109/ISESE.2004.1334896.

[28] G. Wilson *et al.*, "Best practices for scientific computing," *PLOS Biology*, vol. 12, no. 1, pp. 1–7, January 2014. DOI: https://doi.org/10.1371/journal.pbio.1001745.

[29] S. Foote, *Learning to Program*. Addison-Wesley Professional, 2014. [Online]. Available: https://books.google.co.nz/books?id=XHnbBAAAQBAJ (retrieved November 23, 2020).

[30] B. Shneiderman, "Exploratory experiments in programmer behavior," *International Journal of Computer & Information Sciences*, vol. 5, no. 2, pp. 123–143, 1976. DOI: https://doi.org/10.1007/BF00975629.

[31] J. P. Boysen, "Factors affecting computer program comprehension," Ph.D. dissertation, Iowa State University, 1979. DOI: https://doi.org/10.31274/rtd-180813-6131.

[32] J. Siegmund, "Program comprehension: Past, present, and future," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, March 2016, pp. 13–20. DOI: https://doi.org/10.1109/SANER.2016.35.

[33] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: An exploratory study," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 889–903, December 2004. DOI: https://doi.org/10.1109/TSE.2004.101.

[34] GitHub, Inc. "The state of the octoverse," https://octoverse.github.com (retrieved December 12, 2020).

[35] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 345–355. DOI: https://doi.org/10.1145/2568225.2568260.

[36] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: The contributor's perspective," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, ACM SIGSOFT Distinguished paper, Austin, Texas: ACM, May 2016, pp. 285–296. DOI: https://doi.org/10.1145/2884781.2884826.

[37] G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, Florence, Italy: IEEE, 2015, pp. 358–368. DOI: https://doi.org/10.1109/ICSE.2015.55.

[38] S. Chacon, *Pro Git*, ser. Books for professionals by professionals. Apress, 2009. [Online]. Available: https://books.google.co.nz/books?id=3XcW4oJ8goIC (retrieved November 23, 2020).

[39] T. Baum, O. Liskin, K. Niklas, and K. Schneider, "A faceted classification scheme for change-based industrial code review processes," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, August 2016, pp. 74–85. DOI: https://doi.org/10.1109/QRS.2016.19.

[40] M. Fagan, "Design and code inspections to reduce errors in program development," in *Software Pioneers: Contributions to Software Engineering*, M. Broy and E. Denert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 575–607. DOI: https://doi.org/10.1007/978-3-642-59412-0_35.

[41] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 712–721. DOI: https://doi.org/10.1109/ICSE.2013.6606617.

[42] Y. Tymchuk, A. Mocci, and M. Lanza, "Code review: Veni, vidi, vici," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 151–160. DOI: https://doi.org/10.1109/SANER.2015.7081825.

[43] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, Saint Petersburg, Russia: Association for Computing Machinery, 2013, pp. 202–212. DOI: https://doi.org/10.1145/2491411.2491444.

[44] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. "O'Reilly Media, Inc.", 2009.

[45] Y. Goldberg, "A primer on neural network models for natural language processing," *Journal of Artificial Intelligence Research*, vol. 57, pp. 345–420, 2016.

[46] W. Maalej and H. Nabil, "Bug report, feature request, or simply praise? on automatically classifying app reviews," in *2015 IEEE 23rd International Requirements Engineering Conference (RE)*, August 2015, pp. 116–125. DOI: https://doi.org/10.1109/RE.2015.7320414.

[47] E. Guzman, M. Ibrahim, and M. Glinz, "A little bird told me: Mining tweets for requirements and software evolution," in *2017 IEEE 25th International Requirements Engineering Conference (RE)*, Sep. 2017, pp. 11–20. DOI: https://doi.org/10.1109/RE.2017.88.

[48] M.-A. Storey, "Theories, methods and tools in program comprehension: Past, present and future," in *13th International Workshop on Program Comprehension (IWPC'05)*, May 2005, pp. 181–191. DOI: https://doi.org/10.1109/WPC.2005.38.

[49] A. von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, August 1995. DOI: https://doi.org/10.1109/2.402076.

[50] J. Belmonte, P. Dugerdil, and A. Agrawal, "A three-layer model of source code comprehension," in *Proceedings of the 7th India Software Engineering Conference*, ser. ISEC '14, Chennai, India: Association for Computing Machinery, 2014. DOI: https://doi.org/10.1145/2590748.2590758.

[51] T. M. Shaft and I. Vessey, "Research report: The relevance of application domain knowledge: The case of computer program comprehension," *Information Systems Research*, vol. 6, no. 3, pp. 286–299, 1995. [Online]. Available: http://www.jstor.org/stable/23010878 (retrieved November 23, 2020).

[52] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive Psychology*, vol. 19, no. 3, pp. 295 –341, 1987. DOI: https://doi.org/10.1016/0010-0285(87)90007-7.

[53] A. von Mayrhauser and A. M. Vans, "From program comprehension to tool requirements for an industrial environment," in *[1993] IEEE Second Workshop on Program Comprehension*, Jul. 1993, pp. 78–86. DOI: https://doi.org/10.1109/WPC.1993.263903.

[54] F. Détienne and F. Bott, *Software Design – Cognitive Aspect*, ser. Practitioner Series. Springer London, 2002. [Online]. Available: https://link.springer.com/book/10.1007/978-1-4471-0111-6 (retrieved November 23, 2020).

[55] R. Brooks, "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies*, vol. 18, no. 6, pp. 543 –554, 1983. DOI: https://doi.org/10.1016/S0020-7373(83)80031-5.

[56] G. K. Gill and C. F. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *IEEE Transactions on Software Engineering*, vol. 17, no. 12, pp. 1284–1288, December 1991. DOI: https://doi.org/10.1109/32.106988.

[57] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp. 595–609, Sep. 1984. DOI: https://doi.org/10.1109/TSE.1984.5010283.

[58] N. Peitek, J. Siegmund, C. Parnin, S. Apel, J. C. Hofmeister, and A. Brechmann, "Simultaneous measurement of program comprehension with fMRI and eye tracking: A case study," 2018.

[59] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger, "Using psycho-physiological measures to assess task difficulty in software development," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 402–413. DOI: https://doi.org/10.1145/2568225.2568266.

[60] I. Herraiz and A. E. Hassan, *Beyond lines of code: Do we need more complexity metrics?* A. Oram and G. Wilson, Eds. O'Reilly Media, 2010, pp. 125–141. [Online]. Available: https://books.google.co.nz/books?id=DxuGi5h2-HEC (retrieved November 23, 2020).

[61] S. Ajami, Y. Woodbridge, and D. G. Feitelson, "Syntax, predicates, idioms: What really affects code complexity?" In *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC '17, Buenos Aires, Argentina: IEEE Press, 2017, pp. 66–76. DOI: https://doi.org/10.1109/ICPC.2017.39.

[62] S. Yu and S. Zhou, "A survey on metric of software complexity," in *2010 2nd IEEE International Conference on Information Management and Engineering*, April 2010, pp. 352–356. DOI: https://doi.org/10.1109/ICIME.2010.5477581.

[63] M. H. Halstead, *Elements of software science*. Elsevier New York, 1977, vol. 7.

[64] Jingqiu Shao and Yingxu Wang, "A new measure of software complexity based on cognitive weights," *Canadian Journal of Electrical and Computer Engineering*, vol. 28, no. 2, pp. 69–74, April 2003. DOI: https://doi.org/10.1109/CJECE.2003.1532511.

[65] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, December 1976. DOI: https://doi.org/10.1109/TSE.1976.233837.

[66] D. Gopstein *et al.*, "Understanding misunderstandings in source code," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, Paderborn, Germany: ACM, 2017, pp. 129–139. DOI: https://doi.org/10.1145/3106237.3106264.

[67] B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems (Winthrop Computer Systems Series)*. Winthrop Publishers, 1980.

[68] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and Software*, vol. 7, no. 4, pp. 325 –339, 1987. DOI: https://doi.org/10.1016/0164-1212(87)90032-X.

[69] W. L. Johnson and A. Erdem, "Interactive explanation of software systems," *Automated Software Engineering*, vol. 4, no. 1, pp. 53–75, January 1997. DOI: https://doi.org/10.1023/A:1008655629091.

[70] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14, Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 23–34. DOI: https://doi.org/10.1145/1181775.1181779.

[71] T. Fritz and G. C. Murphy, "Using information fragments to answer the questions developers ask," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10, Cape Town, South Africa: ACM, 2010, pp. 175–184. DOI: https://doi.org/10.1145/1806799.1806828.

[72] K. Erdos and H. M. Sneed, "Partial comprehension of complex programs (enough to perform maintenance)," in *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, Jun. 1998, pp. 98–105. DOI: https://doi.org/10.1109/WPC.1998.693322.

[73] A. Stolcke *et al.*, "Dialogue act modeling for automatic tagging and recognition of conversational speech," *Computational Linguistics*, vol. 26, no. 3, pp. 339–373, 2000. DOI: https://doi.org/10.1162/089120100561737.

[74] T. Wu, F. M. Khan, T. A. Fisher, L. A. Shuler, and W. M. Pottenger, "Posting act tagging using transformation-based learning," in *Foundations of Data Mining and knowledge Discovery*, T. Young Lin, S. Ohsuga, C.-J. Liau, X. Hu, and S. Tsumoto, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 319–331. DOI: https://doi.org/10.1007/11498186_18.

[75] E. N. Forsythand and C. H. Martell, "Lexical and discourse analysis of online chat dialog," in *International Conference on Semantic Computing (ICSC 2007)*, Sep. 2007, pp. 19–26. DOI: https://doi.org/10.1109/ICSC.2007.55.

[76] G. Gousios, "The GHTorrent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 233–236. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487132 (retrieved November 23, 2020).

[77] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining GitHub," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, October 2016. DOI: https://doi.org/10.1007/s10664-015-9393-5.

[78] D. Sites, A. Hayden, I. Giuliani, and jariesa, *Compact language detector 2*. [Online]. Available: https://github.com/CLD2Owners/cld2 (retrieved December 4, 2020).

[79] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[80] M. Abadi *et al.* (2015). "TensorFlow: Large-scale machine learning on heterogeneous systems." Software available from tensorflow.org, https://www.tensorflow.org/ (retrieved November 23, 2020).

[81] L. Buitinck *et al.*, "API design for machine learning software: Experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[82] D. Freelon, "ReCal OIR: Ordinal, interval, and ratio intercoder reliability as a web service.," *International Journal of Internet Science*, vol. 8, no. 1, 2013.

[83] K. Krippendorff, *Content Analysis: An Introduction to Its Methodology*. SAGE Publications, 2018. [Online]. Available: https://books.google.co.nz/books?id=FixGDwAAQBAJ (retrieved November 23, 2020).

[84] A. Zheng and A. Casari, *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. O'Reilly Media, 2018. [Online]. Available: https://ebookcentral.proquest.com/lib/auckland/detail.action?docID=5328406 (retrieved November 23, 2020).

[85] K. Pearson, "X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 50, no. 302, pp. 157–175, 1900. DOI: https://doi.org/10.1080/14786440009463897.

[86] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947. DOI: https://doi.org/10.1214/aoms/1177730491.

[87] C. Silva and B. Ribeiro, "The importance of stop word removal on recall values in text categorization," in *Proceedings of the International Joint Conference on Neural Networks, 2003.*, vol. 3, Jul. 2003, 1661–1666 vol.3. DOI: https://doi.org/10.1109/IJCNN.2003.1223656.

[88] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian network classifiers," *Machine Learning*, vol. 29, no. 2, pp. 131–163, November 1997. DOI: https://doi.org/10.1023/A:1007465528199.

[89] A. McCallum, K. Nigam, *et al.*, "A comparison of event models for naive Bayes text classification," in *AAAI-98 workshop on learning for text categorization*, Citeseer, vol. 752, 1998, pp. 41–48.

[90] T. Zhang and F. J. Oles, "Text categorization based on regularized linear classification methods," *Information Retrieval*, vol. 4, no. 1, pp. 5–31, April 2001. DOI: https://doi.org/10.1023/A:1011441423217.

[91] Z. Zheng, X. Wu, and R. Srihari, "Feature selection for text categorization on imbalanced data," *SIGKDD Explor. Newsl.*, vol. 6, no. 1, pp. 80–89, 2004. DOI: https://doi.org/10.1145/1007730.1007741.

[92] T. Joachims, "Text categorization with support vector machines: Learning with many relevant features," in *Machine Learning: ECML-98*, C. Nédellec and C. Rouveirol, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 137–142. DOI: https://doi.org/10.1007/BFb0026683.

[93] N. Jardine and C. van Rijsbergen, "The use of hierarchic clustering in information retrieval," *Information Storage and Retrieval*, vol. 7, no. 5, pp. 217 –240, 1971. DOI: https://doi.org/10.1016/0020-0271(71)90051-9.

[94] C. J. van Rijsbergen, *Information Retrieval*, 2nd. USA: Butterworth-Heinemann, 1979, ch. 7, pp. 112–140. [Online]. Available: http://www.dcs.gla.ac.uk/Keith/Preface.html (retrieved November 23, 2020).

[95] L. Derczynski, "Complementarity, F-score, and NLP evaluation," in *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, 2016, pp. 261–266.

[96] C. Erlingsson and P. Brysiewicz, "A hands-on guide to doing content analysis," *African Journal of Emergency Medicine*, vol. 7, no. 3, pp. 93–99, 2017. DOI: https://doi.org/10.1016/j.afjem.2017.08.001.

[97] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in GitHub: Transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, ser. CSCW '12, Seattle, Washington, USA: Association for Computing Machinery, 2012, pp. 1277–1286. DOI: https://doi.org/10.1145/2145204.2145396.