UNIVERSITÀ DEGLI STUDI DEL MOLISE

DIPARTIMENTO DI BIOSCIENZE E TERRITORIO

Corso di Laurea in
INFORMATICA

Tesi in
AUTOMATED SOFTWARE DELIVERY

A Fine-Grained Analysis of Comments Quality in Code-Related Datasets

Candidato:
Angelo TROTTA
Matricola: 171874

Relatore:
Prof. Simone SCALABRINO

Co-Relatore:
Dr. Antonio VITALE

Anno Accademico 2023/2024

Contents

List of Figures

Chapter 1

Introduction

1.1 Application Context

Large language models (LLMs) have emerged as powerful tools in the field of software development, capable of performing a wide range of coding tasks with impressive accuracy. From generating code snippets and auto-completing functions to identifying bugs and suggesting optimizations, LLMs have become fundamental in streamlining the software development process. They are now being employed in integrated development environments (IDEs), facilitating rapid prototyping, code refactoring, and even explaining code behavior in natural language. However, the quality of code generated by LLMs is highly dependent on the datasets used to train these models. LLMs learn to write code by analyzing vast repositories of source code, comments, and documentation available across different programming languages. If the training data contains high-quality, well-structured code with meaningful comments, the models are more likely to generate precise, readable, and efficient code. On the other hand, poor-quality training data containing incomplete, incorrect, or poorly commented code can lead to less reliable outputs. While comments do not affect the program's execution, they play an essential role in this context [15] [20], critical to software development as a whole in both small teams and large organizations.

Comments are non-executable statements embedded within code to provide explanations, clarify intent, and guide future developers or collaborators who may interact with the codebase. When writing comments, at first glance, general principles and good practices [4] are often overlooked. First of all, it is important to strike a balance in their length, ensuring they are neither too brief nor overly detailed. Comments should be concise yet informative, providing enough context without overwhelming the reader with unnecessary details. For documentation comments, developers must adhere to the specific syntax and formatting rules of the programming language being used, particularly when it comes to annotations and tags, to ensure proper integration with documentation tools. Technical comments added during development should be revised or removed when they become obsolete. Outdated or irrelevant comments can lead to confusion and misinterpretation. Additionally, comments should avoid posing questions, as their primary function is to clarify and

guide, not to introduce uncertainty. Lastly, comments must not be left empty, as they serve no purpose and only add clutter to the codebase.
Given these premises, it is therefore imperative to make sure comments are verified to be correctly written, with as high quality as possible.

1.2 Motivation and Objectives

Comments play an important role in improving the performance of code completion models [5]. The presence of multi-line comments was found to significantly enhance the performance of pre-trained language models like *UniXcoder*, *CodeGPT*, and *InCoder*. *Van Dam et al.* suggest that natural language descriptions embedded in multi-line comments contribute to the models' ability to understand and complete code, making these comments valuable in datasets used for training and evaluating code-related tasks.
Comments are also important in the context of code summarization tasks [21]. High-quality comments enhance the ability of code summarization models to generate accurate and useful descriptions of source code. However, it is also noted how noisy or poor-quality comments can significantly reduce the effectiveness of these models. Given their importance in model training for code-tasks, it was necessary and important to perform an in-depth study, taking on this challenge. Developing a comment analysis system [8] is not an easy task, as it requires a deep understanding of the rules governing the programming language in which the code is written, alongside ensuring computational accuracy and efficiency when processing comment data.
Additionally, the content of comments can often be complex or ambiguous, making it difficult for automated heuristics to handle certain scenarios correctly. In such cases, manual review may be necessary to address edge cases or interpret unclear comments. Therefore, it is crucial to ensure that the system is robust and effective for the vast majority of use cases, while acknowledging that some instances may require additional intervention.

1.3 Results

The first result is a tool to analyze the quality of comments in a given dataset. We have applied NLP techniques in some cases to enhance the quality of the output, conducting an empirical study to evaluate the effectiveness of our approach on sample datasets consisting of manually selected instances.
We then compared the automated results produced by our tool with a thorough manual analysis of the same sample, whose size ensured an error rate within a ±5% margin. After analyzing the aforementioned datasets, we discovered, for instance, that a significant portion of comments are excessively long, and approximately 20% of documentation comments are left incomplete. Finally, we identified the key limitations of our approach and provided recommendations for future research and

improvements. The main contributions of this thesis can be summarized as follows: (I) we proposed an analysis tool for automatic detection of lack of comments quality, evaluated under different criteria, (II) we used a set of carefully selected datasets to train our approach, (III) we defined case-study samples from selected datasets to verify the correctness of our approach.

1.4 Thesis Structure and Organization

The next chapters of this thesis are:

- *Chapter 2*: presents background and related works.
- *Chapter 3*: presents our tool for detecting lack of comments quality in each category and the heuristics used.
- *Chapter 4*: presents the empirical study conducted to validate the defined heuristic.
- *Chapter 5*: concludes this thesis and provides directions for future works.

Chapter 2

Background and Related Work

2.1 Comments Quality

Comment quality refers to how effectively developers convey information about the tasks the code fulfills [2]. Writing clear, helpful comments is essential for maintaining software over time [7]. Source code without adequate comments becomes difficult to maintain, as developers may struggle to make changes without fully understanding its purpose or functionality. High-quality comments should be readable, well-organized, syntactically accurate, and logically clear. They must cater to both experienced and inexperienced developers, ensuring that the code is accessible to anyone who works on it.
There are several practices that can undermine or improve comments quality [6]. For example, when dealing with long and complex methods, the accompanying comment should be concise, straightforward, and focus on key points. Providing examples where relevant can enhance clarity. Comments should also adhere to proper indentation and line-break styles, as inconsistent formatting or excessive spacing can cause confusion and reduce readability.
Issues can arise when outdated or deprecated methods are left commented out within the code [17] [24], often cluttering more useful, current comments. If a comment is intended as documentation, it should match the method's parameters and return types to ensure accuracy and relevance.
Ideally, comments should only be added when necessary. When included, they should provide a description that is both concise and cohesive, avoiding redundancy or excessive detail [14]. Inline comments within methods that explain actions as they occur are particularly useful, as they help guide readers through the code step by step, making it easier to follow and understand.

2.2 Comments Quality Analysis

Several studies have been conducted to analyze the quality of comments across various datasets. One, presented by *Khamis et al.* [12], offers an effective and automated approach for evaluating the quality of inline documentation using a set of heuristics. Their method assesses both the linguistic quality of comments and the consistency

between source code and its corresponding documentation. The researchers applied their approach to different modules of two open-source applications, *ArgoUML* and *Eclipse*, comparing the analysis results with bug reports from the individual modules. This comparison aimed to uncover potential connections between documentation quality and overall code quality, shedding light on how well-maintained comments may influence software reliability.

Steidl et al. [22] proposed a model for evaluating comment quality based on different comment categories. The researchers used machine learning techniques to categorize comments from *Java* and *C/C++* programs, allowing for a more structured analysis of comment quality. By applying metrics tailored to specific comment categories, they were able to effectively assess key quality aspects within the model, providing valuable insights into the strengths and weaknesses of code comments in these programming languages.

Tan Lin [23] highlighted the wealth of information contained in code comments and how this can be used to enhance software maintainability and reliability. He conducted a detailed study and analysis of both free-form and semistructured comments, demonstrating their potential to improve the overall quality and robustness of software systems.

Pooja Rani [18] addressed the problem of evaluating code comments, exploring comment quality assessment from three perspectives: developer concerns, information types, and research support, identifying a taxonomy of common concerns and challenges faced by developers when writing or generating code comments, as well as creating a table of comment quality attributes and metrics, helping to develop a unified approach for evaluating the quality of code comments based on empirical data.

He Hao [9] investigated the patterns and practices of code commenting across different programming languages and project types. By analyzing 150 popular projects in five programming languages (*JavaScript*, *Java*, *C++*, *Python*, and *Go*), the study reveals significant variations in comment density that may be influenced by the programming language and the project's purpose.

Pascarella et al. [16] developed a comprehensive taxonomy for categorizing Java code comments, consisting of six top-level categories and 16 subcategories, validated through extensive manual classification of over 40,000 lines of comments from 14 Java projects, both open-source and industrial. They successfully applied supervised machine learning techniques to automatically classify code comments based on their taxonomy, achieving promising results with precision and recall rates consistently above 93% in most categories for open-source projects, although slightly lower for industrial projects, finding that the most common comment categories were those summarizing code functionality (e.g., "SUMMARY") and providing usage instructions (e.g., "USAGE"). Comments related to development tasks or temporary changes (e.g., "TODO") were less frequent, especially in industrial projects.

Chen et al. [3] developed a machine learning-based method using Random Forests to automatically detect the scope of comments in Java code. Their method achieved an

accuracy of 81.45% in detecting the scopes of comments across four popular open-source projects. This marked a significant improvement over heuristic-based methods, which had an accuracy of 67.10%. The proposed method was applied to two specific tasks: outdated comment detection and automatic comment generation. In both cases, the method improved the performance of baseline approaches, demonstrating its effectiveness in enhancing software repository mining tasks.

Rani et al.[19] presented a comprehensive analysis of research on code comment quality from 2011 to 2020. The study selected and reviewed 47 relevant papers to address four key questions: (I) which types of comments researchers focus on, (II) the quality attributes (QAs) considered, (III) the tools and techniques used to evaluate comment quality, and (IV) the methods used to assess the effectiveness of these studies.

Chapter 3

Analyzer Overview

3.1 Theoretical Construction

The development of our tool followed a structured, multi-step approach. As outlined in *Chapter 1*, to assess comment quality, it was necessary to categorize comments based on specific criteria, described in the figure below, from the work by *Vitale et al.* [25].

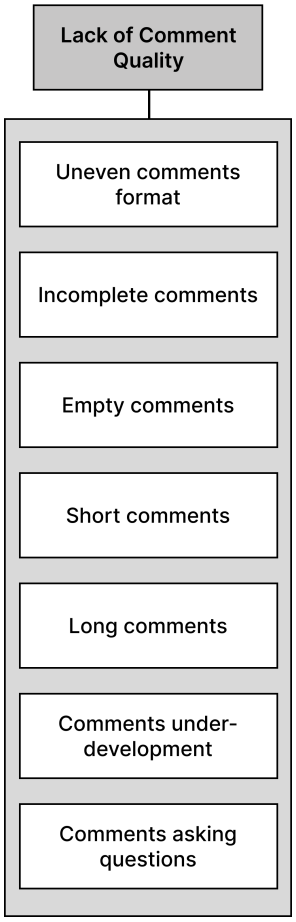

FIGURE 3.1: Categories of comments our study focuses on.

- **Empty comments**: these provide no value or clarity and should be flagged for immediate attention.

- **Comments asking questions**: whether direct or implied, questions undermine the purpose of comments, which should provide clear and definitive information.
- **Short comments**: these fail to offer sufficient context or detail, leaving readers without a clear understanding of the code functionality or purpose.
- **Long comments**: excessively detailed comments can overwhelm readers, leading to confusion and obscuring the main point of the explanation.
- **Comments under development**: these temporary, technical comments are relevant only to active developers and must be regularly updated or removed as the system evolves.
- **Incomplete comments**: these lack full explanations, leaving gaps in the intended message, making them unclear or difficult to understand.
- **Uneven comments format**: inconsistent formatting or structure hinders readability, making it harder to follow and maintain the code. Comments should be clear, consistent, and well-structured to enhance usability.

We began with simpler detections, such as identifying empty comments and comments that posed questions. Established rules were applied to determine if a comment was empty or contained either direct or implied questions.
Next, we tackled short and long comments. To achieve this, we utilized NLTK [1], a comprehensive suite for natural language processing. By tokenizing each comment, we measured its length to classify it as either too short or overly long.
For comments under development, we employed pattern recognition techniques. We incorporated technical keywords and common phrases used during development to detect these comments.
The detection of incomplete comments and uneven comments format was refined iteratively in a trial-and-error way, manually verifying results at each step. For incomplete comments, we initially performed a syntactic analysis to ensure the comments adhered to basic rules of English sentence structure. Subsequently, we built a classification system to categorize comments as single-line, multi-line, or documentation. For documentation comments, we verified consistency between the number of parameters in the comment and the corresponding source code, along with matching return types. Additionally, we introduced checks for single-line and multi-line comments to identify nonsensical or "gibberish" content, hence marking those that lacked clarity or coherence as incomplete.
Finally, for detecting uneven formatting, we flagged comments with irregular indentation, spacing, or annotation tag misuse, tailored to the programming language in use. This ensured that comments maintained proper structure, readability, and compliance with language-specific conventions.

3.2 Logic Construction

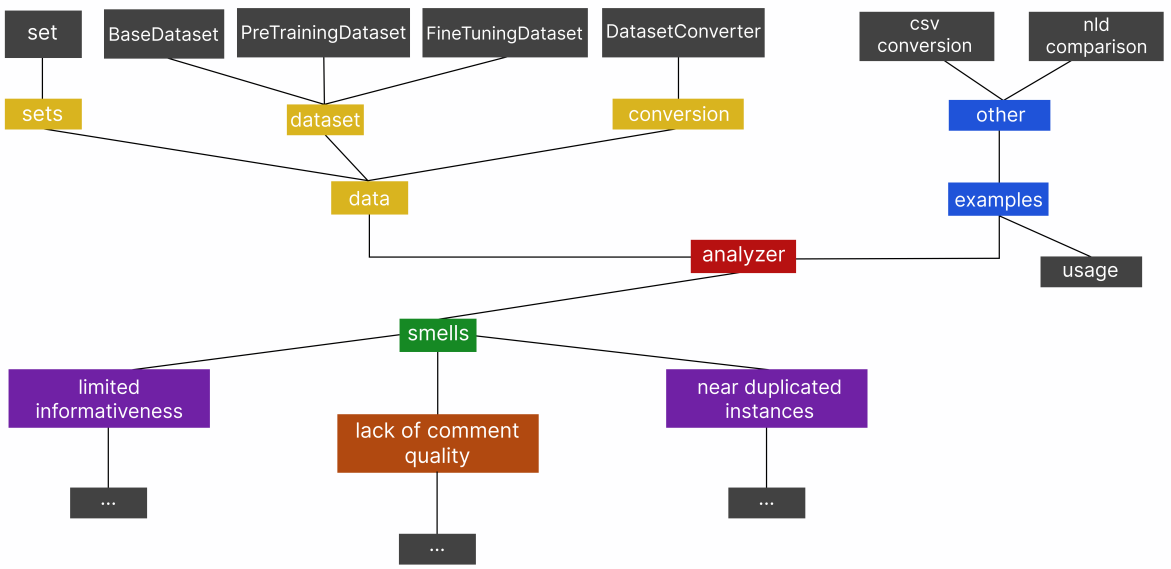

FIGURE 3.2: General view on the tool's architecture.

Our tool's architecture is structured in multiple layers. For clarity, we begin by outlining the foundational components.
The "data" package defines the supported dataset file formats our tool is able to read, provides a common interface for loading and working with datasets, and handles dataset format conversion when necessary.

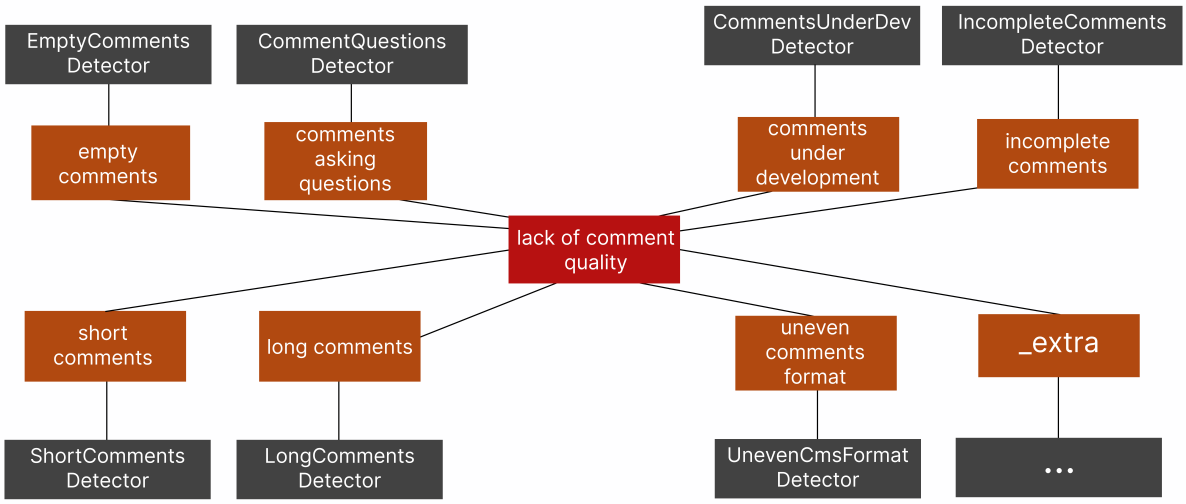

FIGURE 3.3: Zoom on lack of comments quality package.

The "smells" package houses several detectors, but for the purpose of this work, we will focus solely on those that align with the scope of our study. We begin by detailing essential components without which our tool would not function.
Initially, we aimed for the tool to be language-agnostic, meaning it would work with all programming languages. While this is true for some detectors, it is not feasible

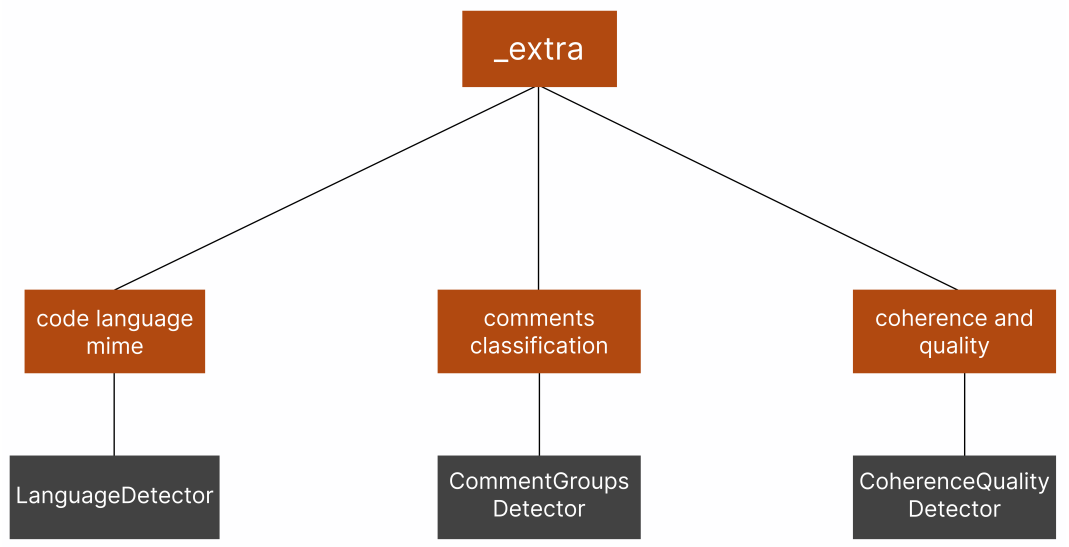

FIGURE 3.4: Zoom on extra sub-package.

for all. As a result, we introduced a programming language detector, represented by the "code_language_mime" sub-package and its corresponding *LanguageDetector* class.
The *CommentGroupsDetector* scans all comments in the input database and categorizes them as single-line, multi-line, or documentation comments. This classification is crucial for enabling the incomplete comments detector and the uneven comments format detector to function correctly.
An additional detector, the *CoherenceQualityDetector*, resides in the "coherence_and_quality" sub-package. This detector was developed while researching comment completeness, but it does not strictly relate to incomplete comments, so it was placed in a separate package. The detector evaluates comments based on the following criteria:

- **Language (L)**: ensures the comment is not written in a mix of different languages.
- **Clarity (C)**: uses NLP techniques to assess the readability of the comment.
- **Relevance (R)**: verifies that the comment accurately describes the specific code constructs it's meant to explain.
- **Brevity (B)**: checks if the comment avoids unnecessary repetition or wordiness.
- **Context (X)**: determines whether the comment provides sufficient explanation for understanding the code.

If a comment meets all these criteria, it is considered coherent. Otherwise, it is flagged as incoherent.

3.3 Heuristics Used

We will list the heuristics for each category related to the lack of comment quality. In all cases, the following notations can be used: let *C* be a code block to be analyzed, let *M* be the MIME type of the code block, let *extract(C, M)* be a method that extracts comments from *C* based on MIME type *M*.
Let *text(i)* represent the text of the i-th comment in the extracted list of comments from *C*.

3.3.1 Empty Comments

Let isEmpty*(i)* be the empty comment condition:

$$\textbf{isEmpty}(i) = \begin{cases} 1, & \text{if strip(text}(i)\text{) = ""} \\ 0, & \text{otherwise} \end{cases}$$

where *strip(text(i))* removes all leading and trailing whitespace from the comment.
We define the function *F(C, M)* that evaluates if the code block *C* contains empty comments based on the MIME type *M*:

$$F(C, M) = \begin{cases} 1, & \text{if } \exists i \text{ such that isEmpty}(i) = 1 \text{ for any comment } i \in \text{extract}(C, M) \\ 0, & \text{otherwise} \end{cases}$$

3.3.2 Comments Asking Questions

Let *K* be a set of interrogative keywords usually found at the start of a sentence.
Let *E* be a set of interrogative expressions that can be found anywhere in a sentence.
Then, we define the condition as isQuestion*(i)*, where:

$$\textbf{isQuestion}(i) = \begin{cases} 1, & \text{if text}(i)\text{.endswith("?")} \vee \exists \text{subset}(K) : \text{text}(i)\text{.startswith}(subset(K)) \\ & \quad \vee \exists \text{subset}(E) \in \text{text}(i) \\ 0, & \text{otherwise} \end{cases}$$

We define the function *F(C, M)* that evaluates if the code block *C* contains any questions based on the MIME type *M*:

$$F(C, M) = \begin{cases} 1, & \text{if } \exists i \text{ such that isQuestion}(i) = 1 \text{ for any comment } i \in \text{extract}(C, M) \\ 0, & \text{otherwise} \end{cases}$$

3.3.3 Short Comments

Let $|\text{T}|_i$ = tokenize(text*(i)*) be the number of tokens in the i-th comment.
Let us define a threshold for short comments as k = 5 (maximum number of tokens).

Let us define the short comments condition as isShort*(i)*, where:

$$\textbf{isShort}(i) = \begin{cases} 1, & \text{if } |\mathrm{T}|_i \leq k \\ 0, & \text{otherwise} \end{cases}$$

where $|\mathrm{T}|_i$ represents the number of tokens in the i-th comment.
We define the function *F(C, M)* that evaluates if the code block *C* contains any short comments based on the MIME type *M*:

$$F(C, M) = \begin{cases} 1, & \text{if } \exists i \text{ such that isShort}(i) = 1 \text{ for any comment } i \in \text{extract}(C, M) \\ 0, & \text{otherwise} \end{cases}$$

3.3.4 Long Comments

The heuristic is similar to that used for short comments, but in the first case, the comparison is reversed. It is inspired by the work of *Steidl et al.* [22], but our approach uses tokens rather than words.
Let us define a threshold for long comments as k = 30 (minimum number of tokens).
Let us define the long comments condition as isLong*(i)*, where:

$$\textbf{isLong}(i) = \begin{cases} 1, & \text{if } |\mathrm{T}|_i \geq k \\ 0, & \text{otherwise} \end{cases}$$

where $|\mathrm{T}|_i$ represents the number of tokens in the i-th comment.

$$F(C, M) = \begin{cases} 1, & \text{if } \exists i \text{ such that isLong}(i) = 1 \text{ for any comment } i \in \text{extract}(C, M) \\ 0, & \text{otherwise} \end{cases}$$

3.3.5 Comments Under Development

This heuristic is inspired by the work of *Shi et al.* [21].
Let us define the following sets of keywords:

- P = placeholder keywords, e.g., "Description of the Method", "Logic goes here".
- N = temporary notes, e.g., "Add more info here", "Temporary fix for issue".
- D = vague descriptions, e.g., "To be documented", "Work in progress".
- R = redundant keys, e.g., "Opens a file", "Calls the method".
- U = URL keys, e.g., "https://", "www.", ".com"
- Let R_M represent a set of MIME-type specific patterns to detect commented methods (e.g., method definitions in Java, Python, etc.).

Let regexMatch(*i, P, N, D, R, U,* R_M) be a function that checks whether any of the patterns defined by *P, N, D, R, U,* R_M match the comment text *text(i)*.
Let us define the under development condition as isUnderDev*(i)*, where:

$$\textbf{isUnderDev}(i) = \begin{cases} 1, & \text{if regexMatch(i, P, N, D, R, U, } R_M) = 1 \\ 0, & \text{otherwise} \end{cases}$$

We define the function *F(C, M)* that evaluates if the code block *C* contains any comments indicative of being "under development" based on the MIME type *M*:

$$F(C, M) = \begin{cases} 1, & \text{if } \exists i \text{ such that isUnderDev}(i) = 1 \text{ for any comment } i \in \text{extract}(C, M) \\ 0, & \text{otherwise} \end{cases}$$

3.3.6 Incomplete Comments

Let *SML* represent groups of comments classified by *CommentGroupsDetector* into single-line or multi-line.
Let *DOC* represent the group of comments classified as documentation by *CommentGroupsDetector*.
Let us assume we are able to detect different types of comment formats (e.g., *Google*, *NumPy*, *ReST*, *epytext*, *Javadoc*) based on the detected structure of comments and the programming language.
For single-line and multi-line comments, first we have to check whether a comment is written in English.

$$\mathrm{F}_{english}(i) = \begin{cases} 1, & \text{if comment } i \in SML \ \wedge \ \text{text}(i) \text{ detected as English} \\ 0, & \text{if comment } i \in SML \ \wedge \ \text{text}(i) \text{ NOT detected as English} \end{cases}$$

Then, we check the syntactic completeness of a comment by verifying if each sentence contains a subject and a predicate, assuming the comment $\in$ *SML*.

$$\mathrm{F}_{syntax}(i) = \begin{cases} 1, & \text{if } \exists \text{ sentence } \in \text{ comment } i \text{ syntactically incomplete} \\ 0, & \text{if all sentences are complete} \end{cases}$$

Lastly, we evaluate whether a comment, assumed $\in$ *SML*, is gibberish using a pretrained model [11] that classifies comments into various categories.

$$\mathrm{F}_{gibberish}(i) = \begin{cases} 1, & \text{if } \exists \text{ sentence } \in \text{ comment } i \text{ classified as gibberish} \\ 0, & \text{if all sentences are clean} \end{cases}$$

For documentation comments, we must check if a comment is missing any critical part in annotation tags (wrong number of params, lack of return type) compared to the code block it refers to.

Let P_i represent the number of *@param* annotations in the comment, assuming comment $\in$ *DOC*.
Let R_i represent the number of *@return* annotations in the comment (which can be only 0 or 1), assuming comment $\in$ *DOC*.
We check the number of parameters in the function signature.

$$F_{params}(C) = \begin{cases} n, & \text{if the function has parameters} \\ 0, & \text{if no parameters exist} \end{cases}$$

Then, we check the return type in the function signature.

$$F_{return}(C) = \begin{cases} 1, & \text{if it returns a value other than void} \\ 0, & \text{if it returns void or no return type is found} \end{cases}$$

To conclude we can say that, for a single-line or multi-line comment to be incomplete, the following conditions must be met:

1. The comment is in English.
2. The comment is syntactically incomplete.
3. The comment contains gibberish.

Thus, the overall incompleteness function for the i-th single-line or multi-line comment $F^{SML}_{incomplete}$ can be expressed as:

$$F^{SML}_{incomplete}(i) = \begin{cases} 1, & \text{if } F_{english}(i) = 1 \wedge F_{syntax}(i) = 1 \wedge F_{gibberish}(i) = 1 \\ 0, & \text{otherwise} \end{cases}$$

For a documentation comment to be incomplete, it means there is either a mismatch between the number of *@param* annotations and the actual number of parameters, or a mismatch between the presence of a *@return* annotation and whether the function actually returns a value.
This implies that the incompleteness function for the i-th documentation comment in the code block *C*, $F^{DOC}_{incomplete}$ can be expressed as:

$$F^{DOC}_{incomplete}(C, i) = \begin{cases} 1, & \text{if } P_i \neq F_{params}(C) \\ 1, & \text{if } R_i = 1 \wedge F_{return}(C) = 0 \\ 1, & \text{if } R_i = 0 \wedge F_{return}(C) = 1 \\ 0, & \text{otherwise} \end{cases}$$

3.3.7 Uneven Comments Format

Let *SL* be the group of single-line comments, *ML* the group of multi-line comments and *DOC* the group of documentation comments.

Let *SF* be the single-line way of comment formatting (e.g, // in *Java*, # in *Python*).
Let *MF* be the multi-line way of comment formatting (e.g, /* or /** in *Java*, ''' or """ in *Python*)
Let *TAGS* be an array of commonly used HTML tags in Javadoc comments.
Let *S* be a sentence $\in$ comment.
Let spaces*(S)* represent the number of spaces in that sentence.
Let *K* be the maximum number of spaces allowed per sentence.
We have a function that checks whether there are too many spaces (more than *K* consecutive) in any sentence from the i-th comment, which could indicate indentation issues.

$$\mathrm{F}_{spacing}(i) = \begin{cases} 1, & \text{if } \exists S \in \text{comment } i \ : \ \text{spaces}(S) > K \\ 0, & \text{otherwise} \end{cases}$$

Single-line and multi-line comments should not contain any HTML tags, as they are treated as plain text. Assuming comment $\in SL \vee$ comment $\in ML$:

$$\text{hasTags}(i) = \begin{cases} 1, & \text{if } \exists T \subset TAGS : T \in \text{comment } i \\ 0, & \text{otherwise} \end{cases}$$

For the i-th single-line comment to be uneven, we check the number of spaces or if it matches the multi-line format.

$$\mathrm{F}^{SL}_{uneven}(i, MF) = \begin{cases} 1, & \text{if } \mathrm{F}_{spacing}(i) = 1 \vee \text{hasTags}(i) = 1 \vee match(i, MF) \\ 0, & \text{otherwise} \end{cases}$$

For the i-th multi-line comment to be uneven, we do the opposite.

$$\mathrm{F}^{ML}_{uneven}(i, SF) = \begin{cases} 1, & \text{if } \mathrm{F}_{spacing}(i) = 1 \vee \text{hasTags}(i) = 1 \vee match(i, SF) \\ 0, & \text{otherwise} \end{cases}$$

For documentation comments, we have a function that checks for irregularities in the use of annotation tags (e.g, wrong indentation, empty annotation, wrong token after annotation, ends with annotation). In particular, if the annotation is *@author* and the following token is a verb, it makes no sense. If the annotation is followed by a redundant token (e.g, @return nothing), it should be avoided.

Let *RDT* be an array of possible redundant words after an annotation (e.g, author, param, nothing, none, void). Assume i-th comment $\in$ *DOC*, and the current token is an annotation.

$$\mathrm{F}_{uneven}^{DOC}(i) = \begin{cases} 1, & \text{if token} = \textit{'@author'} \wedge \textit{token.next} \in \textit{'VERB'} \\ 1, & \text{if isSpace}(\textit{token.next}) = 1 \wedge \mathrm{F}_{\textit{spacing}}(i) = 1 \\ 1, & \text{if isEmpty}(\textit{token.next}) = 1 \\ 1, & \text{if } \textit{token.next} \in \textit{RDT} \\ 0, & \text{otherwise} \end{cases}$$

Chapter 4

Empirical Study

In this chapter, we present an empirical study aimed at validating the effectiveness of the tool that was developed. We want to create a highly precise system capable of detecting previously outlined deficiencies in code comments, ensuring accurate identification of such.

4.1 Study Design

The goal of this study is to understand to what extent the proposed heuristics allow to detect true comments lacking quality. Our study is steered by the following research question:

RQ: How accurate are our heuristics in detecting poor-quality comment smells in state-of-the-art datasets?

4.1.1 Data Collection

In order to validate our heuristic, we focused on: *CodeSearchNet* (CSN) [10] and *Automatically Generating Review Comments with Pre-training Models* (AUGER) [13] datasets, both vastly used in literature, selecting from each two random samples of 384 instances. This number is important as a sample of this size guarantees ±5% margin of error and 95% confidence level.
CSN is a large-scale dataset designed to facilitate semantic code search, which is the process of retrieving relevant code snippets based on natural language queries. The dataset includes over 6 million functions from six programming languages, but our subset focuses solely on *Java*.
AUGER is designed for generating automated review comments in code reviews, using the Text-to-Text Transfer Transformer (T5) pre-trained model. The dataset includes data collected from 11 notable Java open-source projects on GitHub.

4.1.2 Data Analysis

We manually analyzed comments from the sample, labeling each with a value of 0 when the comment was not meeting a detection criteria, and 1 otherwise. In ambiguous cases, we defaulted to 0. For the purpose of validating our heuristic, we

compared its results against our manual annotations, using the metrics of accuracy, precision, recall, and f1score for each criteria. These metrics were computed with the following formulas:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

$$f1score = 2 \times \frac{precision \times recall}{precision + recall}$$

With the term TP we indicate those comments that are caught by our heuristics in one of the detection criteria. With the term TN we indicate those comments that are not caught by our heuristics in one or any of the detection criteria. With the term FP we indicate those comments that are detected by our heuristic in one of the criteria but actually should not. With the term FN we indicate those comments that are not detected by our heuristic in one or any of the criteria but actually should.

4.2 Study Results

4.2.1 CSN

After carefully reviewing the sample multiple times, we arrived at the following results:

	Accuracy	**Precision**	**Recall**	**F1 Score**
Comments Classification*	97%	99%	95%	97%
Empty Comments	97%	63%	67%	65%
Comments Asking Questions	99%	60%	75%	67%
Short Comments	97%	88%	98%	93%
Long Comments	90%	89%	92%	91%
Comments Under Development	96%	83%	58%	68%
Incomplete Comments	86%	94%	89%	92%
Uneven Comments Format	87%	93%	83%	88%

**Comments Classification* refers to the detector that is able to scan all comments and categorize them in single-line, multi-line and documentation. See paragraph 3.2.

From the table above, it can be deduced that our heuristics generally perform well, showing strong metrics in most cases. However, there are some notable caveats. The results for *Empty Comments*, *Comments Asking Questions*, and *Comments Under Development* are visibly less balanced, with lower precision, recall, and F1 scores. This is primarily due to the small number of true positives in these categories, which has skewed the metrics.

To better understand this issue, we can refer to the confusion matrix values for these categories:

	TP	**TN**	**FP**	**FN**
Empty Comments	10	362	6	5
Comments Asking Questions	3	378	2	1
Comments Under Development	15	355	3	11

As we can see, the number of true positives (TP) is relatively low in these cases, which significantly impacts the precision, recall, and F1 score. For example, only 3 true positives were identified for *Comments Asking Questions*, which contributes to the other low scores despite high accuracy. Meanwhile, *Comments Under Development* shows a relatively high number of false negatives (FP) which affects recall and F1 score despite high accuracy and precision.

4.2.2 AUGER

Repeating the same process as previously applied for *CSN*, we arrived at the following results:

	Accuracy	**Precision**	**Recall**	**F1 Score**
Comments Classification	97%	98%	95%	97%
Empty Comments	99%	63%	71%	67%
Comments Asking Questions	97%	96%	95%	95%
Short Comments	92%	81%	90%	85%
Long Comments	94%	95%	92%	93%
Comments Under Development	77%	78%	61%	69%
Incomplete Comments	84%	94%	73%	82%
Uneven Comments Format	77%	81%	61%	70%

Overall, we achieved similar results across both datasets, though with some differences. Since *AUGER* is a dataset specifically for code review comments, it contained a lot more instances of comments asking questions and comments under development compared to *CSN*. This contributed to higher detection rates in both categories. However, only in the case of *Comments Under Development*, our manual analysis identified more instances than the tool was able to detect, resulting in more positive results for *Comments Asking Questions* and more negative results for *Comments Under Development* regarding accuracy and precision compared to *CSN*. Additionally, *AUGER* had fewer examples of unevenly formatted comments, leading to lower scores in that area. These variations highlight the differing nature of the datasets and their influence on the tool's performance across specific categories.

4.3 Threats to Validity

This section discusses the threats to validity of this heuristic. Specifically, (I) construction validity regarding manual testing, (II) internal validity regarding possible impairment of results and (III) external validity regarding the dataset.

4.3.1 Construct Validity

We have detailed the logic behind our heuristics and described the validation process used to evaluate their effectiveness by extracting key metrics. The heuristics were tested on 384 instances from the datasets of *CSN* and *AUGER*. This process involved manually labeling each comment, meaning that any misinterpretation of the comments could potentially compromise the validity of the results. To mitigate this risk, we thoroughly reviewed and examined all comments multiple times to ensure accuracy and reduce the likelihood of errors in interpretation.

4.3.2 Internal Validity

To address the research question, we developed heuristics grounded in our knowledge of the domain. This approach involved incorporating our own insights and patterns related to comment analysis, which may have introduced some subjectivity, particularly in the more complex detectors, such as those for incomplete comments and uneven comment formatting. However, we aimed to select rules that are broadly applicable and objective, while maintaining a disciplined approach to minimize subjectivity and ensure consistency in our criteria.
We acknowledge that with different samples, results might vary, potentially yielding more balanced or imbalanced outcomes. However, the current analysis provides valuable insights into where improvements could be made in future iterations.

4.3.3 External Validity

We selected small, random samples from very large datasets, which resulted in analyzing only a limited number of instances relative to the dataset size. This sampling approach may have led to some imbalances in the results. However, our primary goal was to design heuristics that minimized false positives and false negatives. Although we tested these heuristics across a variety of comment categories, certain categories had very few relevant instances, suggesting that the results could be influenced by the specific sample chosen and may not fully generalize to the entire dataset. Additionally, factors such as the programming language used and the presence of external frameworks, which often fall outside the built-in syntax and conventions of the language, could further affect the generalizability of our findings. These factors may limit the ability to extend our results to other datasets or environments without further validation.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

We explored the critical role of code comment analysis in improving software maintainability, readability, and overall quality. Many analyses have been made to study the quality of comments, yet the majority of them focus on key specifics, one or two study scopes. We aim for more general approaches that allow to study comments quality from a variety of points of view. As such, we presented several criteria that help us divide comments in different categories, and for each category we have developed an appropriate heuristic. By leveraging natural language processing (NLP) and machine learning techniques, we evaluated the tool effectiveness on state-of-the-art datasets, finding average accuracy and precision scores respectively of 94% and 84% for *CSN*, 90% and 86% for *AUGER*.

5.2 Future Work

This work has laid the foundation for numerous future advancements in code comment analysis. The categorizations and heuristics developed can serve as valuable resources for building machine learning models that enhance detection accuracy and extend the analysis beyond the initial scope. Future research can explore additional categories and aspects of comments that can be inferred, while also expanding the tool's applicability to a wider range of programming languages.
One of our ongoing goals is to further minimize false positives by refining the heuristics to be more precise and objective, ensuring that as few relevant instances as possible are excluded. Ultimately, this work contributes to the expanding field of automated quality assurance, helping to bridge the gap between human-readable comments and machine-driven analysis, and paving the way for more maintainable and robust codebases in the future.

Bibliography

[1] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[2] Jürgen Börstler et al. "Developers talking about code quality". In: *Empirical Software Engineering* 28.6 (2023), p. 128. ISSN: 1573-7616. DOI: 10.1007/s10664-023-10381-0. URL: https://doi.org/10.1007/s10664-023-10381-0.

[3] Huanchao Chen et al. "Automatically detecting the scopes of source code comments". In: *Journal of Systems and Software* 153 (2019), pp. 45–63. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.03.010. URL: https://www.sciencedirect.com/science/article/pii/S016412121930055X.

[4] Diana Chien and Paul Clemons. "Coding and Comment Style". In: *MIT Communication Lab* 1.10 (2019), pp. 3–9. URL: https://mitcommlab.mit.edu/broad/commkit/coding-and-comment-style/.

[5] Tim VAN Dam, Maliheh Izadi, and Arie VAN Deursen. *Enriching Source Code with Contextual Data for Code Completion Models: An Empirical Study*. 2023. arXiv: 2304.12269 [cs.CL]. URL: https://arxiv.org/abs/2304.12269.

[6] Weihao Ding. *Good Comments vs. Bad Comments*. 2022. URL: https://www.directimpactsolutions.com/en/good-comments-vs-bad-comments/.

[7] Beat Fluri, Michael Wursch, and Harald C. Gall. "Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes". In: *14th Working Conference on Reverse Engineering (WCRE 2007)*. 2007, pp. 70–79. DOI: 10.1109/WCRE.2007.21.

[8] Syed Zohaib Hassan et al. "Code Comment Analysis–A Review Paper". In: *Journal of Management Practices, Humanities and Social Sciences* 6.1 (2022), pp. 88–105. DOI: 10.33152/jmphss-6.1.9. URL: https://global-jws.com/ojs/index.php/global-jws/article/view/126.

[9] Hao He. "Understanding source code comments at large-scale". In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. Ed. by Marlon Dumas et al. ACM, 2019, pp. 1217–1219. DOI: 10.1145/3338906.3342494. URL: https://doi.org/10.1145/3338906.3342494.

[10] Hamel Husain et al. *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search*. 2020. arXiv: 1909.09436 [cs.LG]. URL: https://arxiv.org/abs/1909.09436.

[11] Madhur Jindal. *autonlp-Gibberish-Detector-492513457 (Revision 34458f9)*. 2024. DOI: 10.57967/hf/2664. URL: https://huggingface.co/madhurjindal/autonlp-Gibberish-Detector-492513457.

[12] Ninus Khamis, René Witte, and Juergen Rilling. "Automatic Quality Assessment of Source Code Comments: The JavadocMiner". In: *Natural Language Processing and Information Systems*. Ed. by Christina J Hopfe et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 68–79.

[13] Lingwei Li et al. "AUGER: automatically generating review comments with pre-training models". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, 1009–1021. ISBN: 9781450394130. DOI: 10.1145/3540250.3549099. URL: https://doi.org/10.1145/3540250.3549099.

[14] Annie Louis et al. *Deep Learning to Detect Redundant Method Comments*. 2018. arXiv: 1806.04616 [cs.SE]. URL: https://arxiv.org/abs/1806.04616.

[15] Delano Oliveira et al. "Evaluating Code Readability and Legibility: An Examination of Human-centric Studies". In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020, pp. 348–359. DOI: 10.1109/ICSME46990.2020.00041.

[16] Bruntink M. & Bacchelli Pascarella L. "Classifying code comments in Java software systems". In: *Empir Software Eng* 1.24 (2019), pp. 1499–1537.

[17] Shiva Radmanesh et al. *Investigating the Impact of Code Comment Inconsistency on Bug Introducing*. 2024. arXiv: 2409.10781 [cs.SE]. URL: https://arxiv.org/abs/2409.10781.

[18] Pooja Rani. "Speculative Analysis for Quality Assessment of Code Comments". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. Vol. 36. IEEE, May 2021, 299–303. DOI: 10.1109/icse-companion52605.2021.00132. URL: http://dx.doi.org/10.1109/ICSE-Companion52605.2021.00132.

[19] Pooja Rani et al. "A decade of code comment quality assessment: A systematic literature review". In: *Journal of Systems and Software* 195 (2023), p. 111515. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2022.111515. URL: https://www.sciencedirect.com/science/article/pii/S0164121222001911.

[20] Berna Seref and Ozgur Tanriover. "Software Code Maintainability : A Literature Review". In: *International Journal of Software Engineering and Applications* 7 (May 2016), pp. 69–87. DOI: 10.5121/ijsea.2016.7305.

[21] Lin Shi et al. *Are We Building on the Rock? On the Importance of Data Preprocessing for Code Summarization*. 2022. arXiv: 2207.05579 [cs.SE]. URL: https://arxiv.org/abs/2207.05579.

[22] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. "Quality analysis of source code comments". In: *2013 21st International Conference on Program Comprehension (ICPC)*. 2013, pp. 83–92. DOI: 10.1109/ICPC.2013.6613836.

[23] Lin Tan. "Chapter 17 - Code Comment Analysis for Improving Software Quality**This chapter contains figures, tables, and text copied from the author's PhD dissertation and the papers that the author of this chapter coauthored [[3], [1], [35], [7]]. Sections 17.2.3, 17.4.3, 17.5, and 17.6 are new, and the other sections are augmented, reorganized, and improved." In: *The Art and Science of Analyzing Software Data*. Ed. by Christian Bird, Tim Menzies, and Thomas Zimmermann. Boston: Morgan Kaufmann, 2015, pp. 493–517. ISBN: 978-0-12-411519-4. DOI: https://doi.org/10.1016/B978-0-12-411519-4.00017-3. URL: https://www.sciencedirect.com/science/article/pii/B9780124115194000173.

[24] Wen Siang Tan, Markus Wagner, and Christoph Treude. *Detecting Outdated Code Element References in Software Repository Documentation*. 2022. arXiv: 2212.01479 [cs.SE]. URL: https://arxiv.org/abs/2212.01479.

[25] Antonio Vitale, Rocco Oliveto, and Simone Scalabrino. *A Catalog of Data Smells for Coding Tasks*.