Bachelor's Thesis

Static Code Smell Analysis Using Large Language Models: An Empirical Study with Llama 3

Liubomyr Hromadiuk

December 9, 2024

Advisor:
Dr. Norman Peitek Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel Chair of Software Engineering
Prof. Dr. Vera Demberg Chair of Computer Science and Computational Linguistics

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT DES SAARLANDES

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

Abstract

Even well-functioning code often contains hidden code smells - indicators of possible issues that hinder software quality. Undetected code smells result in maintenance problems, increasing technical debt. This study explores the potential use of Llama 3, the state-of-the-art Large Language Model (LLM), in detection of different code smells. Our findings demonstrate that Llama 3 shows competitive performance, particularly in detecting structural code smells; however, further statistical analysis indicates no significant differences in overall performance between the LLM and other static analyzers ,i.e, PMD, Checkstyle and SonarQube. These results suggest that Llama 3 is not yet ready to replace static analysis tools entirely , but can serve as a valuable complementary tool, sparing a significant amount of programmers' time.

Contents

List of Figures

List of Tables

Listings

Acronyms

GenAI Generative AI

LLM Large Language Model

MRR Mean Reciprocal Rank

SRP Single Responsibility Principle

1 Introduction

Recently, there has been a significant increase in the number of people learning how to code. More young individuals are interested in this field, lowering the age frames of those entering the programming world [36]. In today's digital era, programming skills are becoming more common and are no longer limited to computer science graduates. This leads to a growing number of potentially inexperienced developers who collaborate on diverse software projects. Consequently, the ability to write clean, understandable, and maintainable code has gained importance. Supporting this viewpoint, McConnell further explored this matter, discussing software construction practices that address maintainability concerns [26]. Chen et al. [4] have revealed shortcomings that arise from neglecting maintenance-related issues. Addressing this problem, Naseef et al. demonstrated [22] that code smells pose a substantial threat and are a common issue that results in poor maintainability and reduced code quality.

Although modern methods of detecting and fixing code smells have been proving their effectiveness since the beginning of the century, they either are not capturing more complex relationships between code elements and are lacking understanding of the overall context of the system or are highly resource-intensive [12, 24]. For example, some machine learning models, although achieving high accuracy, tend to be computationally expensive, as they need to train a separate model for each type of code smell [16, 45].

Recent advancements of Large Language Models (LLMs) can not be overseen. They have proven their usefulness in various domains, including code generation, machine translation, question answering, speech recognition, and have become an unavoidable helper in the daily routine of millions of people [13]. The latest advances in the field of Generative AI (GenAI) have led to significant contributions of LLMs in software development world. The quantitative findings of Oliver Bodemer demonstrate significant improvements in efficiency and error reduction, underscoring the value of integrating AI tools into the development process [2]. A simple prompt, written in natural language, allows LLMs to produce complex code in a matter of seconds, speeding up all stages of the software development process, including coding and testing [9].

Using LLMs in code smell detection can be a revolutionary alternative to existing analysis tools. As noted above, they can process both natural and programming languages, meaning that they can effectively understand complex relationships within code that might not be processed by other tools of analysis. This insight is useful in detecting defects that require further analysis of the system. Moreover, due to the versatility of LLMs, they can learn from vast amounts of data. This makes them potentially better at identifying new and unnamed smells and even giving simple refactoring solutions for fixing them.

1.1 Goal of This Thesis

The principal aim of this thesis is to investigate and assess the potential of using LLMs, on the example of Llama 3, to detect different code smells in source code. In particular, we are interested in how well LLMs can identify code smells compared to already established practices, i.e., CheckStyle, SonarQube, PMD, which are further discussed in the following chapter.

Despite the advancements described above, code quality issues are still prevalent in AI-generated code [21]. To support this claim, we refer to the study conducted by Yetiştiren et al., which reveals that the latest versions of ChatGPT, GitHub Copilot, and Amazon CodeWhisperer generate correct code 65.2%, 46.3%, and 31.1% of the time, respectively [54]. Thus, we are mainly interested in whether LLMs can be used to mitigate code quality issues rather than generate new code from scratch.

Given this objective, we hypothesize that LLMs perform at least as well as existing code smell detection tools and that they may even significantly outperform current approaches as they address the drawbacks of the latter. If this assumption holds, this in turn will contribute to the development of more efficient and effective methods for code smell detection, which can ultimately improve software quality, reduce maintenance costs, and enhance developer productivity.

1.2 Overview

This thesis is structured into five further chapters. Chapter 2 presents a comprehensive overview of the theoretical fundamentals necessary to understand and follow the content of this thesis, along with an analysis of related work. The proceeding Chapter 3 introduces research questions, metrics for further evaluation, as well as the detailed research procedure. Chapter 4 discusses the results acquired and addresses research questions. Chapter 5 considers possible limitations and threats to validity. The concluding Chapter 6 provides a summary of this thesis and highlights ideas for future work.

2

Background and Related Work

This chapter provides fundamental background information to understand and follow the contents of this thesis. We offer a comprehensive overview of code smells with a particular focus on five specific types relevant for this thesis. Subsequently, we also introduce static code smell analyzers and LLMs.

2.1 Code Smells

Modern approaches in computer science frequently find origins in nature and utilize these understandings to address complex problems and create revolutionary technologies. In the natural world, smells help organisms communicate and evaluate each other [18]. For example, animals rely on them to mark their territories, find suitable partners, and detect potential dangers within their habitats. Flowers, following the same principle, release distinct fragrances to attract, warn, or even repel other species. Scents in these scenarios reveal hidden signals that help others understand their surroundings and make informed choices.

Analogously, in the software engineering domain, code smells serve a similar role. They are indicators of potential issues in the source code that may hinder software maintenance and readability [49]. Fowler [8] highlighted 22 different code smells and introduced respective refactoring strategies. Rahman et al. [40] have shown that some code smells tend to occur more frequently than others: *Long Parameter List* and *Complex Method* were proven to be among the most common ones. In order to provide a better insight into how code smells can affect software development, this section will describe five different types in more detail. They have been selected to show the majority of problems that programmers can encounter and to present specific refactoring approaches that can help solve them. The selection of these code smells is based on their recognized impact on different aspects of software design and prevalence in real-world projects [30, 39, 48].

2.1.1 Long Parameter List

Long Parameter List [25] is a code smell characterized by a method or function that has excessively many parameters, making the code complex to comprehend and extend. Such methods usually expect the callers to provide a lot of context information, making the method prone to errors. The problem escalates when the method has a number of parameters of the same type, which can easily lead to confusion and wrong results. Figure 2.1 introduces

an example of such a method, along with the possible refactoring strategy that involves the encapsulation of the parameters in a separate class.

```
public void printPerson(String firstName,
                        String lastName,
                        String middleName,
                        String maidenName,
                        String nickName,
                        int age,
                        int height) {
  ...
}
```

```
public void printPerson(Person person) {
  ...
}
```

Figure 2.1: Long Parameter List: Code smell (left) and refactored version (right)

2.1.2 Complex Method

While an excessive amount of parameters overcomplicate the interface of the method, resulting in deterioration of software design and posing numerous difficulties during its utilization, the complexity of the function itself may additionally cause severe maintainability concerns. It hinders the overall comprehension of the program, and even requires expensive refactoring during the latter stages of the development process.

Complex Method [48] is detected to be present if the code is cognitively complex, long, handles many responsibilities, or has difficult control structures, including deep nesting or multiple loops. The latter can result in code duplication or hidden dependencies. One of the commonly used refactoring techniques suggests dividing a large and complicated method into smaller and more specific ones, each of which performs a particular action as shown in Figure 2.2.

```
public void analyzeString(String word){
  // 100 lines of code with numerous
  // nested for-loops and if statements
  ...
}
```

```
public void analyzeString(String word){
  analyzeWordStructure(String word);
  countCharacterOccurrences(String word);
  detectPartOfSpeech(String word);
  lemmatizeWord(String word);
}
```

Figure 2.2: Complex method: Code smell (left) and refactored version (right)

2.1.3 Data Class

Data Class code smell suggests that a class is primarily used to store information and has little or no methods that manipulate the data. These classes are usually composed of fields, getters, and setters, but do not contain any meaningful logic. Even though data classes may appear to be quite innocent, they are usually a sign of a bad design and can lead to violations of the principles of object orientation, such as data encapsulation and information hiding. Fowler suggested transferring all the methods and behaviors associated with the

data to the data class, which will improve the cohesion of the class and minimize external coupling in the program [8].

2.1.4 Multifaceted Abstraction

Multifaceted Abstraction refers to a situation in which an abstraction manages several, frequently unrelated, issues or functionalities. The multiple responsibilities issue infringes the Single Responsibility Principle (SRP). Martin [25] has argued that violating the SRP results in low cohesion and high coupling, which inevitably increases technical debt. For example, `java.sql.DataTruncation` class from Figure 2.3 can act either as a warning or an exception based on the function it is used in. This approach violates the principle described above and could lead to confusion and errors, as warnings and exceptions are handled and caught differently in the settings of this example. A possible solution to the problem would be to create two separate classes for each of the functionalities as shown on the right-hand side of Figure 2.3.

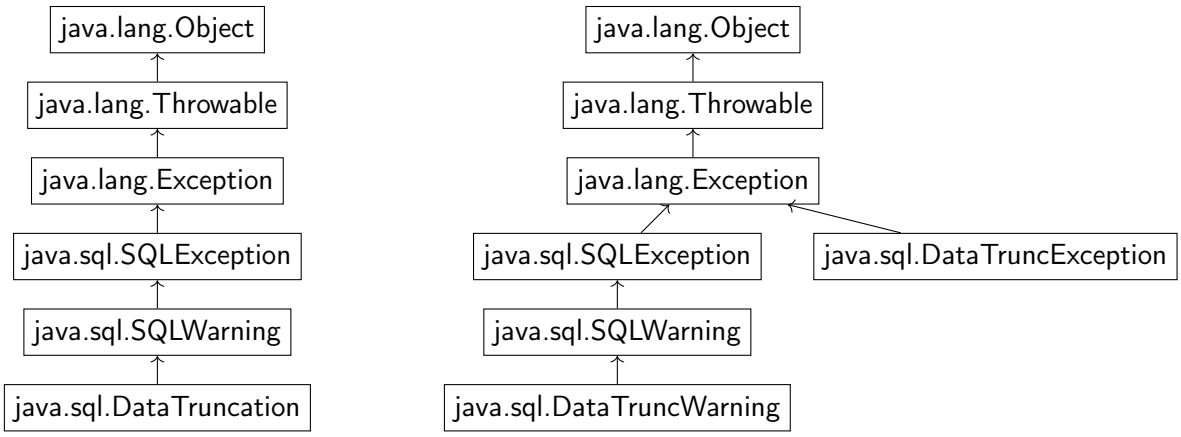

Figure 2.3: Multifaceted Abstraction: Simplified Class Diagram [35]: The problematic design (left) and refactored design (right)

2.1.5 Feature Envy

Feature Envy is a code smell that occurs when a method in one class appears to be profoundly interested in the data or behavior of another class. The method is in the wrong place, because instead of mainly working with its own data, it interacts with and modifies the data of an external class. This makes the code less modular and more difficult to manage, as changes to the data structure or behavior of one class can impact others. To address this problem, the method should ideally be relocated to the class of its primary interest, which will enhance the encapsulation. Figure 2.4 illustrates an example scenario of *Feature Envy*. The `Transaction` class contains the method `applyFee()`. However, this method needs and modifies the data of the `Account` class only. The current misplacement creates unnecessary coupling between the `Transaction` and `Account` classes, complicating maintenance. Instead

of having an extra class, it would prove beneficial to refactor the method into `Account` class. By performing this change, the design adheres to the principle of encapsulation and eliminates the potential errors or overhead resulting from anticipating any changes in case `Account` class is modified.

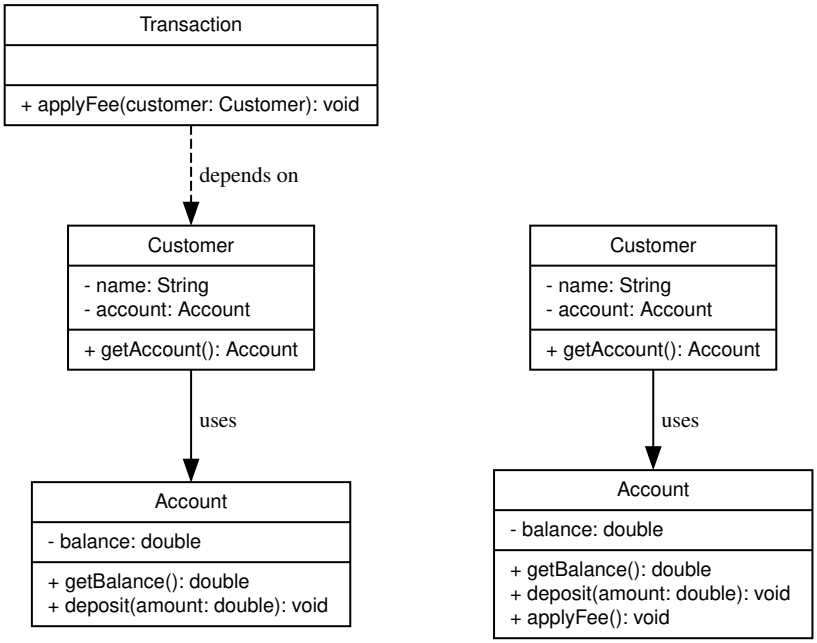

Figure 2.4: Feature Envy UML Diagram: The problematic design (left) and refactored design (right).

2.2 Code Smell Analysis

We introduced several code smells in the previous section, our goal is to be able to detect them which leads us into the field of code smell analysis. Code smell analysis has been an active research area in software engineering for several decades [19], with its inception marked in 1999 when Fowler and Beck [8] introduced and coined the term *code smell*. Shortly afterward, researchers and programmers started identifying patterns that would signal the presence of undesired artifacts. The first methods were based on manual inspection using simple rules and code reviews, which were time-consuming and prone to human errors [15]. To address these limitations, in the mid-2000s automated, metric-based methods were developed [3, 28, 37]. Most of the tools calculate different measurements, such as Cyclomatic Complexity, Lack of Cohesion of Methods, Depth of Inheritance, among others. The corresponding indicators are computed and compared to predefined thresholds. However, these thresholds are not uniformly defined and rely on the specific data and interpretation of the programmer [38].

With the advancements in the machine learning field, researchers searched for the opportunity to benefit from new approaches in order to improve performance and ensure wider applicability. Azeem [1] pointed out that computer scientists have tried to apply machine learning algorithms and models to address the limitations of metric-based approaches. For

example, Sharma et al. [45] employed a transfer-learning technique to detect code smells, demonstrating the potential of learning-based approaches in this domain. Specifically, they concluded that both convolutional and recurrent neural networks, as well as autoencoder deep learning models can be used for code smell detection, although their performance highly depends on a specific type of code smell and thus they do not always outperform other metric-based analyzers. Another prominent example was introduced by Hall et al. [14], who utilized Random Forest as their core concept. Having solved the problem of understanding the context, the machine learning techniques still required experts to perform feature extraction and leave room for performance and energy consumption improvements. Even though, DeepSmells, 1D-CNN (Convolutional Neural Network) model, has reached state-of-the-art performance, although outperforming other machine learning models, trains a different model for each code smell, which is not ideal [16]. The evolution of code smell analysis is summarized in Figure 2.5.

Nowadays there have been several efforts to use LLMs to analyze source code, detect bugs, and code smells, since they are endowed with natural language processing [41]. Most of these efforts have been directed towards assessing how effectively the well-known model, namely ChatGPT, can perform this task [46, 47], while excluding a comparative analysis of other LLMs and their possible efficiency in identifying code smells and software defects. The application of LLMs to code smell detection offers a promising alternative to traditional static analysis tools.

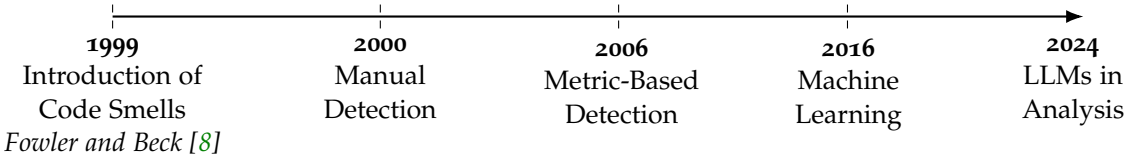

Figure 2.5: Key milestones in the evolution of code smell analysis: A simplified timeline

2.2.1 Static Analyzers

Static code smell analysis is the process of detecting code smells in the source code without executing the program. While a plethora of static analyzers exist, we examine three of the most popular ones in this study according to Yeboah et al.[53]: PMD, Checkstyle, and SonarQube.

2.2.1.1 *PMD*

PMD is an extensible multilanguage static code analyzer [37]. It finds common programming flaws like unused variables, empty catch blocks, and unnecessary object creation, just to name a few. It is mainly concerned with Java and Apex, but supports 16 other languages. It operates in a manner that is very similar to conventional static analysis tools. This naturally means that the tool involves the generation and traversal of an abstract syntax tree [32].

PMD has been shown to be effective in large-scale software projects due to its extensibility and ability to be integrated with various development environments [53]. Although it

has been observed that PMD can effectively identify simple code smells, it may not be as effective in identifying complex patterns or smells on higher abstraction levels that require the understanding of the code semantics [7, 17].

2.2.1.2 *Checkstyle*

Checkstyle [3] is a static code analysis tool designed primarily to enforce coding standards and conventions in Java applications. The tool employs a set of rules that are either predefined or defined by the user to look for problems like absence of Javadoc comment, non-adherence to common practices and standards or incorrect syntax of code snippets. Due to its compatibility with build tools such as Apache Maven, Gradle, and continuous integration environments, it is ideal for large-scale projects to maintain code quality.

2.2.1.3 *SonarQube*

SonarQube [50] is an open source solution for code quality inspection that supports most of the programming languages such as Java, C, C++, Python, and JavaScript. It inspects source code to identify a broad range of quality defects including bugs, security flaws, code smells, and duplications, and provides all the results in a web interface. Recent studies [29] highlight SonarQube's ability to detect a broad spectrum of code issues and its effectiveness in improving software maintainability.

2.3 Large Language Models

In the scope of this thesis, we aim to investigate whether Large Language Models are a suitable alternative to static analyzers discussed above. Large Language Models are a subset of AI models built on deep learning. They process large amounts of data, trying to comprehend complex statistical dependencies and generate human-like text. In order to be able to predict billions of parameters they try to understand statistical dependencies between words and sentences during a self-supervised and semi-supervised training process. Large Language Models appear in various forms and manners with their unique strengths and weaknesses. While some of them, such as ChatGPT [33], outperform others in question answering tasks, others have mastered multilingual understanding (e.g., PaLM 2 [11]), code completion (e.g., GitHub Copilot [10]) or even advanced content generation tasks, such as code (e.g., Codex [5]) or image generation (e.g., DALL-E [34]). All of these and many more models describe different ways of how LLMs have already enriched the field of AI research and development. They have not only demonstrated the potential to automate complex tasks but have also provided innovation across industries, from healthcare and education to entertainment and software engineering.

2.3.1 Llama 3

The third generation of LLMs created by Meta is known as Llama 3 [51]. By adding a larger and more diverse training set along with more sophisticated training methods, and

utilization of Grouped-Query Attention this version improves upon its predecessors in terms of understanding and producing texts that resemble human-generated ones. A greater understanding of language and its surrounding context is made possible by the model's architectural changes, which include an increased number of parameters and improved procedures. Notably, it uses a tokenizer with a vocabulary of 128.000 tokens, enabling more efficient language encoding and improved model performance [51].

Many factors motivate the choice to focus on Llama 3 in this study, while many available LLMs could have been selected instead. Llama 3 shows impressive capabilities in many natural language processing tasks and it excels at activities such as code completion, text generation and information retrieval [44]. Strong performance in tasks related to code makes this model a good fit for code smell analysis compared, for example, to well-known image-generative DALL-E [34]. Furthermore, due to its open-source nature, users can understand the reasoning behind some decisions made, which could be helpful for tasks involving the prompt engineering domain [6]. Llama 3 offers a balance between advanced features, accessibility, and computational bearability, making it a more practical choice for research purposes compared to models like ChatGPT.

In this chapter, we introduced LLMs, certain types of code smells along with brief history of code smell analysis. Code smells remain an obstacle that hinders the maintenance of the programs. Existing static analyzers are not ideal and leave room for further improvements. Considering recent advancements in the field of GenAI, LLMs seem to be a good alternative. In the following chapter, we describe our research design.

3

Methodology

This chapter presents the research questions, outlines the general methodology, and details the procedures employed to ensure systematical and consistent research.

3.1 Research Questions

In our research, we address the following questions:

> **RQ1** What types of code smells are most accurately detected by Llama 3?

Even though Llama 3 might generally perform well, its accuracy could vary depending on the type of a specific code smell. Therefore, we are particularly interested not only in whether Llama 3 is able to correctly detect code smells, but also in what specific code smell from our predefined set it identifies best. This research helps us understand the model's strengths and shortcomings in terms of the subtle nature of distinct code smells, as well as whether its perception is consistent with human interpretations.

> **RQ2** How effective is Llama 3 in detecting and classifying code smells compared to existing static analysis tools?

Despite recent breakthroughs in development of machine learning and GenAI models, the quantitative comparison of LLMs and traditional static analysis approaches utilized for code smell detection is still a venue to explore. In order to provide clearer insights, we compare the performances of Llama 3 and classic analyzers, such as PMD, Checkstyle and SonarQube. We evaluate them based on their precision, recall, and F1-score metrics, using an imbalanced dataset to reflect real-world scenarios. This comparison offers a perspective on whether Llama 3's capabilities are superior, complementary, or inferior to current methods used in industry. This implies that we can assess the potential of Llama 3 to effectively replace or improve other static analysis techniques.

> **RQ3** How accurately does Llama 3 prioritize code smells in terms of relevance, that is at what rank does it correctly identify the most prominent code smell?

Considering the situation when Llama 3 fails to detect the code smell on the first attempt, can it still identify it later on? This question is designed to test how closely the model can

resemble human priorities for code smells, and how well its ranks align with severity of the defect. This evaluation allows us to assess the usefulness of Llama 3 for tasks that involve prioritization, for example, during code review or when deciding on refactoring strategies.

3.2 Operationalization

To answer RQ1 and RQ2, we consider our task to be a multinomial classification problem, meaning that each code smell corresponds to a class. This involves evaluation of model's performance on each class separately, as well as jointly (overall) performance. We employ precision, recall, and F1-score as per class metrics. In addition, we regard each class as having equal importance. Consequently, for evaluation of overall performance macro-averaged metrics are used.

For RQ2, we go a step further to compare the F1-scores of Llama 3 with those of traditional tools, i.e., PMD, Checkstyle, and SonarQube, using statistical testing. In particular, we employ the Kruskal-Wallis test in order to assess whether the differences in F1-scores between the tools are statistically significant. If differences are found to be significant, we use the Wilcoxon Signed-Rank test in order to determine performance of which tools is significantly different. This approach provides a solid statistical basis for analyzing and comparing the overall effectiveness of the tools, which we discuss in further detail below.

Precision

Precision, also known as positive predictive value, is defined as the ratio of true positive observations (TP) to the total number of positive observations ($TP + FP$):

$$\text{Precision}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i},$$

where:

- TP_i: Number of true positive observations for class i (correctly identified instances).
- FP_i: Number of false positive observations for class i (instances incorrectly assigned to class i).

Macro-average precision is the average precision across all classes:

$$\text{Precision}_{\text{Macro}} = \frac{1}{N} \sum_{i=1}^{N} \text{Precision}_i,$$

where:

- N: Total number of classes.

Recall

Recall, also known as sensitivity or the true positive rate, for a class is the ratio of true positives to the sum of true positives and false negatives:

$$\text{Recall}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i},$$

where:

- TP_i: Number of true positive observations for class i.
- FN_i: Number of false negative observations for class i (instances belonging to class i but not identified as such).

Macro-average recall is the average recall across all classes:

$$\text{Recall}_{\text{Macro}} = \frac{1}{N} \sum_i \text{Recall}_i,$$

F1-Score

F1-score is the harmonic mean of precision and recall. This is a comprehensive metric that provides insight into both false-positive and false-negative errors, balancing precision and recall.

$$\text{F1-Score}_i = \frac{2 \cdot \mathit{Precision}_i \cdot \mathit{Recall}_i}{\mathit{Precision}_i + \mathit{Recall}_i},$$

The macro-average F1 score is the average F1-Score across all classes:

$$\text{F1-Score}_{\text{Macro}} = \frac{1}{N} \sum_i \text{F1-Score}_i,$$

Kruskal-Wallis Test

The Kruskal-Wallis test [20] is a non-parametric statistical test which is used to compare more than two independent groups of data and to determine whether there are any significant differences between their medians. It is essentially applicable when the normal distribution and equal variance assumptions of ANOVA and other similar tests are violated. Due to its robustness, it is recommended to routinely use Kruskal–Wallis, unless the data is based on a large sample size and clearly normally distributed [27]. Thus, we use the Kruskal-Wallis test.

The Kruskal-Wallis test statistic H without ties (duplicates) is calculated as follows:

$$H = \frac{12}{N(N+1)} \sum_{i=1}^{c} \frac{R_i^2}{n_i} - 3(N+1),$$

where:

- N: Total number of observations in all groups.
- c: Number of groups being compared.
- n_i: Number of observations in group i.
- R_i: Sum of the ranks in group i.

The test follows a χ^2-distribution with $c-1$ degrees of freedom. The null hypothesis (H_0) suggests that the medians of all groups are equal, i.e., there is no significant difference between them. The alternative hypothesis H_1, on the other hand, presumes that at least one group has a different median. If the p-value is below the significance level, the null hypothesis is rejected, indicating significant differences between the groups.

Wilcoxon Signed-Rank Test

The Wilcoxon signed-rank test [42] is a non-parametric test with principal aim to compare two paired groups and to determine whether their population mean ranks differ. It is often used as a post-hoc test after the Kruskal-Wallis test to compare pairs of groups[27]. In our study, F1-scores for different analysis tools are calculated for the same set of classes. This implies that the data is inherently paired, as the performance for each tool corresponds to the same class.

The Wilcoxon test statistic W is computed as the sum of the ranks of absolute differences:

$$W = \sum_{i=1}^{n} \text{rank}_{|x_i - y_i|} \cdot \text{signum}(x_i - y_i),$$

where:

- x_i, y_i: Paired observations.
- $\text{rank}_{|x_i - y_i|}$: Rank of the absolute difference between paired observations.
- $\text{signum}(x_i - y_i)$: Sign of the difference between paired observations.

The null hypothesis (H_0) states that the median difference between the two groups is zero, thus, they do not differ. The alternative hypothesis (H_1) postulates that the median difference between the two groups deviated from zero.

Mean Reciprocal Rank

For RQ3, we aim to determine at what rank (attempt) Llama 3 correctly identifies the present code smell. As an estimate, we use Mean Reciprocal Rank (MRR). MRR is the average of the reciprocal ranks of the first correct answer. It is calculated as follows:

$$\text{MRR} = \frac{1}{Q} \sum_{i=1}^{Q} \frac{1}{\text{rank}_i},$$

where:

- Q: Total number of code snippets analyzed.
- rank_i: Rank (attempt) of the first correct prediction for snippet i.

3.3 Procedure

To ensure a fair and comprehensive evaluation, our research process is divided into several steps.

Code Smell	Type	Dataset	Count
Complex Method	Method	DACOS	1 340
Data Class	Class	MLCQ	4 020
Feature Envy	Method	MLCQ	3 332
Long Parameter List	Method	DACOS	1 443
Multifaceted Abstraction	Class	DACOS	2 402
None	-	DACOSX	7 463
Total			**20 000**

Table 3.1: Distribution of code smells in the created dataset

Step 0: Dataset construction

We used two datasets: MLCQ [23] and DACOS (combined with DACOSX) [31]. These datasets contain a variety of smelly code snippets from real-world Java projects. All the codes in these datasets have been labeled by programming experts to make sure that the annotations are accurate and reliable. Manual annotation is crucial in identifying and confirming the existence of code smells since it reduces biases and mistakes that may occur during automatic or heuristic-based approach. This is to ensure that the datasets are valid and appropriate for assessing the performance of the proposed detection methods. Furthermore, we used the DACOSX, extension of DACOS, to include benign code snippets. This step ensures that the chosen methods not only correctly handle smelly code, but also do not identify well-written code as potentially harmful. Various metrics together with extreme thresholds were introduced to ensure benign code smells are smell-free, thereby providing a solid background for our research. Due to computational constraints and to ensure practical feasibility and to ease the reproducibility of our findings, we limit the number of code snippets to 20 000, the distribution is shown in Table 3.1.

Step 1: Data Collection and Preprocessing

The MLCQ dataset provides links to GitHub repositories, which we use to extract relevant lines of code as specified in the dataset. The DACOS dataset, on the other hand, provides the code snippets in form of files. However, these files are saved in a format that is different from the standard Java source code which is expected by the static analyzers used in our study. This `.code` format makes the raw snippets incompatible with automated analysis tools meaning that these snippets have to be preprocessed in order to get the correct `.java`

```java
public class ExampleClass {
    // content of the code snippet
}
```

Listing 3.1: Template for wrapping code snippets into a syntactically correct Java class

ending. Furthermore, code snippets that only contain methods must be wrapped in a class to ensure syntactically correctness. We use Listing 3.1 as a template to wrap those code snippets that belong to a method as indicated in Table 3.1.

Step 2: Acquire Predictions

In this thesis we use *Meta-Llama-3-8B-instruct* model. This model has been specifically upgraded for handling instruction-based tasks, thus making it appropriate for dealing with code snippets [51].

The following user prompt addresses RQ1 and RQ2:

```
Analyze the following code snippet and identify the most prominent
code smell according to established software engineering
principles. If there is no discernible code smell, respond with
'None'.
```

The creation of proper prompt is vital during investigating the behavior of LLMs [43]. In this study, we explored different prompt engineering strategies to optimize the instruction making its responses more relevant. Adhering to concepts of directional-stimulus prompting, the model is told to *"identify the most prominent code smell"* or to reply *"None"* if no code smell is detected, this helps in minimizing confusion and enhances the coherency of responses. Principles of generated knowledge prompting postulate that the model should make use of its embedded understanding, we ensure it by explicitly stating: *"according to established software engineering principles"* to provide relevant and domain-specific outputs. Additionally, the prompt has been optimized through the process of iterational fine-tuning: the prompt was modified, depending on the initial responses, to make it clearer, more relevant to the objectives of the research and easier to understand.

We employ a slightly modified prompt for RQ3:

```
Analyze the following code snippet and identify the most prominent
code smells according to established software engineering
principles. If there are no discernible code smells, respond with
'None'. Rank the detected code smells in order of relevance.
```

We performed analysis for PMD v7.3.0, Checkstyle v10.17.0, and SonarQube v10.6 through scripted automation to ensure consistent and efficient analysis. To better reflect the nature of the datasets, minor adjustments to certain rules were made. Mainly, threshold for Cyclomatic Complexity was set to 7, maximal number of allowed parameters was set to 4.

Step 3: Analysis and Interpretation of Results

To determine whether the respective tool has correctly identified the presence or absence of a specific code smell, we use a set of rules and keywords, as summarized in Table A.1.

Unrelated to our research questions, we additionally noticed that Llama 3 can classify a code snippet as belonging to a class of code smell not presented in the originally constructed

dataset. As it is infeasible to verify the correctness of such predictions, we decided to uniformly exclude them from the final evaluation across all analyzer. In order to comply to the principle of fairness, we firstly processed the output of Llama 3. The total number of 1646 samples (approximately 8.23%) were excluded from the final evaluation due to the aforementioned reason. The code snippets for which Llama 3 had provided an incorrect answer were identified and collected. These snippets were then used to create a more specific dataset for RQ3 where the focus was on the model's capability to filter prioritize relevant code smells. The final metrics relevant for evaluation are calculated as described in the previous section.

4

Evaluation

In this chapter, we present the results obtained structured along our three research questions and subsequently interpret and discuss them.

4.1 Results

Following the process described in the previous section, we were able to acquire the following results.

4.1.1 RQ1: Detection Accuracy

The first research question focuses on investigating the overall performance of Llama 3 in terms of code smell detection. Furthermore, we are interested in what specific types of code smells Llama 3 is able to identify most accurately.

Llama 3 demonstrates both strengths and weaknesses in code smell detection. The model is able to correctly identify *Data Class* code smell with a high level of precision (0.7342) and a high recall of 0.9045, giving it a high F1-score of 0.8105. On the contrary, for *Multifaceted Abstraction* and *Feature Envy*, Llama 3 shows the lowest values of precision, recall, and F1-score. Considering smell-free code and *Long Parameter List*, although being highly precise, LLM struggles with recall. In other words, Llama 3 appears to be cautious in identifying these code smells, yet very accurate when it does. This is relatively consistent with the *Complex Method*, it has the precision of 0.7783, recall of 0.6112, and a fairly good F1-score of 0.6847. This means that the model is able to identify this smell rather well but may not get to grasp certain aspects of it. The lowest precision and recall are equal to 0.1215 and 0.0779 respectively and an F1-score of 0.095. Detailed per code smell metrics are summarized in Table 4.1.

RQ1	Overall, Llama 3 has shown a moderate performance across the different code smells, which proves that it is accurate but not always consistent in terms of the recall especially for more complex smells including Multifaceted Abstraction and Feature Envy where the higher levels of abstraction are used and additional data flow analysis is required.

Code Smell	TP	FP	FN	TN	Precision	Recall	F1-Score
Data Class	3190	1155	337	1911	0.7342	0.9045	**0.8105**
Feature Envy	239	1729	2827	1798	0.1215	0.0779	0.0950
Multifaceted Abstraction	232	387	1809	9333	0.3745	0.1135	0.1742
Long Parameter List	507	203	722	10329	0.7142	0.4127	0.5231
Complex Method	744	212	473	10332	0.7783	0.6112	0.6847
None	3162	591	4085	3923	0.8425	0.4363	0.5749

Table 4.1: Overview of the Llama 3 results per code smell. TP: True Positives, FP: False Positives, FN: False Negatives, TN: True Negatives

4.1.2 RQ2: Performance Across Analyzers

The aim of the second research question was to evaluate how effectively Llama 3 identifies and classifies different code smells compared to other static analyzers, i.e., PMD, Checkstyle, and SonarQube.

Table 4.2 shows macro-averaged metrics for each of the analyzers. Llama 3 achieves a precision of 0.5942, which is higher than that of Checkstyle (0.5219), but lower than the precision of PMD (0.6524) and SonarQube (0.6592). Considering recall, with the value of 0.4261 Llama 3 outperforms Checkstyle (0.3329) and PMD (0.3230), but falls short of SonarQube (0.6001). F1-score performance follows the same trend as the recall, meaning that Llama 3 (0.4771) performs better than Checkstyle (0.3953) and PMD (0.3989), but does not reach the level of SonarQube (0.5459). We show detailed results in Table A.2.

Statistical Tests

Figure 4.2 illustrates boxplots of F1-score distributions for each code smell grouped by analyzer. To assess whether the observed differences between medians are statistically significant, we performed the Kruskal-Wallis test. We applied it to the results shown in Figure 4.1. The Kruskal-Wallis test yielded the following results: $H = 0.9533$, $p = 0.8125 > 0.05$. Since the calculated p-value is much larger than the chosen confidence level, we fail to reject the null hypothesis. Thus, we conclude that there is no statistical difference in the overall performance among all analyzers. Further statistical tests, that is, Wilcoxon

Analyzer	Precision	Recall	F1 - Score
Llama 3	0.5942	0.4261	0.4771
Checkstyle	0.5219	0.3329	0.3953
SonarQube	0.6592	0.6001	0.5459
PMD	0.6524	0.3230	0.3989

Table 4.2: Overview of the joint results: Macro-averaged metrics

signed-rank tests, would not add anything substantial to the understanding and may even lead to performing redundant analyses, since if none of the analyzers' performance significantly differs from others, it is also impossible that there exists a pair of analyzers whose performance is statistically unalike.

RQ2 Llama 3 demonstrates competitive performance, particularly in achieving a balanced trade-off between precision and recall. Generally, it outperforms PMD and Checkstyle, but is less effective than SonarQube. However, there is no statistically significant difference between the performance of the tools.

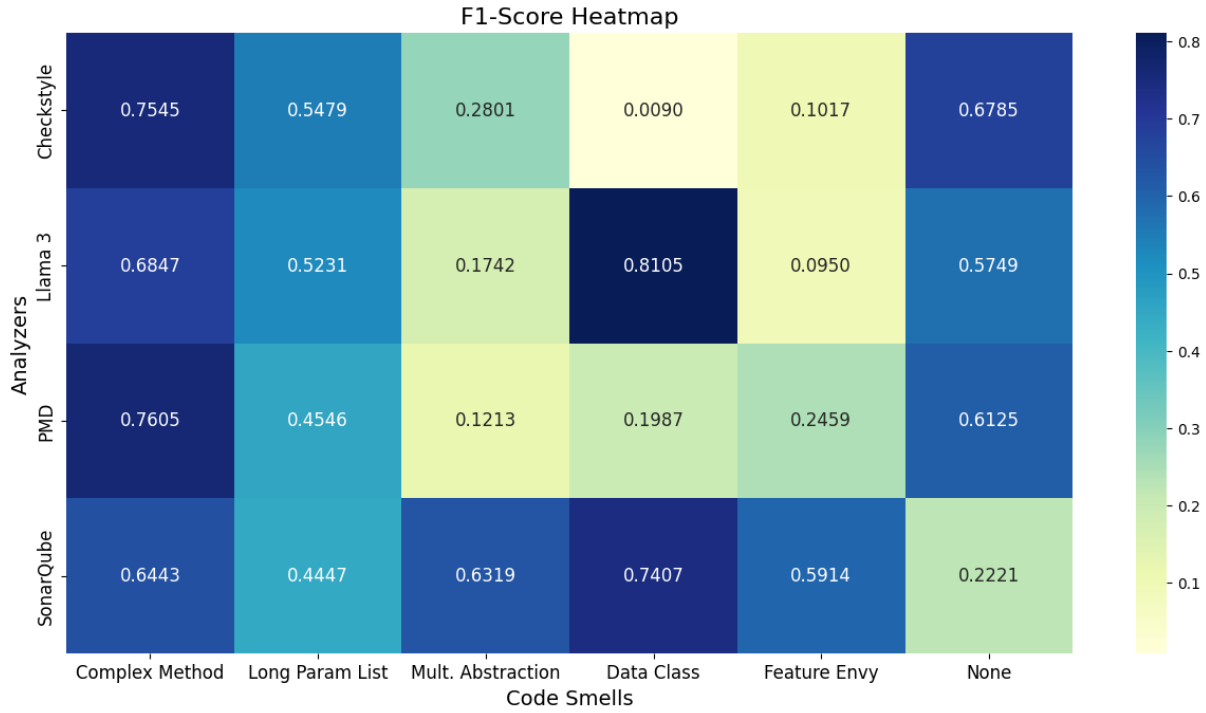

Figure 4.1: Heatmap representing F1-Scores for different code smells across analyzers

4.1.3 RQ3: Ranking Accuracy

The final research question investigated how accurately Llama 3 is able to prioritize code smells in terms of relevance, assessing its alignment with human priorities. The MRR was calculated to be 0.2842, meaning average rank of the first correct prediction was:

$$\text{AvgRank} = \frac{1}{0.2842} \approx 3.5$$

.

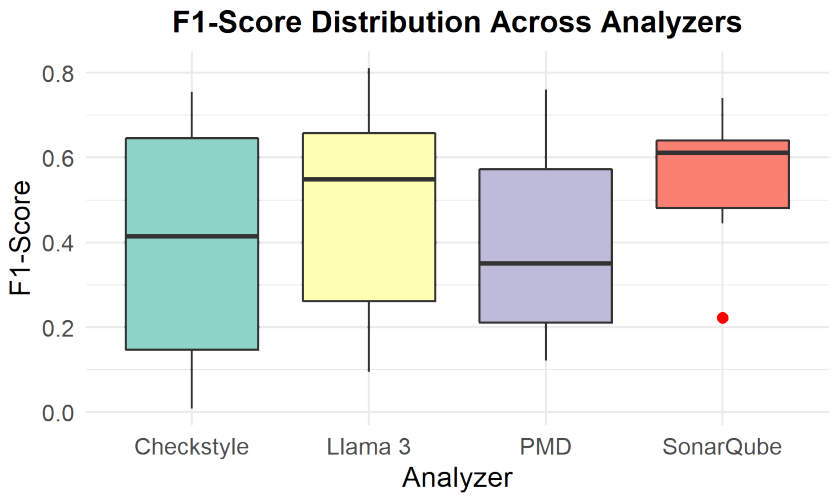

Figure 4.2: Boxplot showing the distribution of F1-scores per code smell for different analysis tools

RQ3 Llama 3 shows modest accuracy in prioritizing code smells, with an MRR of 0.2842, corresponding to an average rank of 3.5 for the first correct prediction.

4.2 Discussion

In this section, we further elaborate on results and discuss possible implications and applications.

4.2.1 RQ1: Detection Accuracy

As the results indicated in the previous section, Llama 3 demonstrated varying performance depending on a specific type of code smell. The model shows impressive results in identifying the *Data class*. This also aligns well with the findings of Fontana et al. [7] who showed that various machine learning models performed best on *Data class* code smell, indicating the similarity in performance between machine learning models and LLMs. However, for *Multifaceted Abstraction* and *Feature Envy*, the effectiveness of Llama 3 drops significantly, highlighting the lowest values of precision, recall, and F1-score. It is worth mentioning that other static analyzers have mostly shown similar unsatisfying results, as illustrated in Table A.2, for these two code smells. It inevitably hints at the fact that for these code smells more surrounding information as well as deeper data flow analysis are required. For other types of smells and benign code, the model shows moderate results, meaning that while being able to identify them, the performance cannot be classified as exceptionally good or bad. We also hypothesize that the results are, to a certain extent, influenced by the different proportions of respective code smell snippets present in the original training set.

From the personal perspective, anticipated high performance for smells like *Data class* and *Complex method* proves the ability of LLMs to identify code smells that follow a specific

pattern. Meanwhile, the limited efficiency for *Multifaceted Abstraction* and *Feauture envy* reflects current limitations and suggests areas of further research.

4.2.2 RQ2: Performance Across Analyzers

The competitive performance of Llama 3 highlights the potential of using LLMs in the field of code smell analysis. Even though the model outperforms Checkstyle and PMD in general, it struggles to beat SonarQube. The statistical evaluation of overall performance suggests that there are no important differences in how well the tools perform. Considering all of the above, we can conclude that LLMs such as Llama 3 may be a suitable supplement to existing solutions; however, their out-of-the-box versions are not powerful enough to substitute static analyzers entirely. This finding supports the viewpoint of Wu et al. [52] who tried to combine metric-based approaches with LLM-based techniques to achieve persuading results in code refactoring.

4.2.3 RQ3: Ranking Accuracy

The MRR of 0.2842 and respective average rank of approximately 3.5 means that, on average, Llama 3 correctly identifies the most relevant code smell at the 3rd or 4th position in its ranked list of predictions. This finding shows that in order to achieve better results, we could tailor Llama 3 to this specific task, performing fine-tuning. Important improvements to its prioritization capabilities could be achieved through the incorporation of ranking algorithms or through the embedding of principles that resemble human comprehension of the program. The correct smell is placed mostly within the top four ranks and its promising ability suggests that, with refinement, it could be used as a helpful tool for developers in tasks of prioritization and refactoring.

5 Threats to Validity

Threats to validity are critical to address in any research to ensure the reliability and applicability of the findings. In this section, we distinguish between internal and external validity and elaborate on specific issues pertinent to the scope of this thesis.

5.1 Internal Validity

One significant threat to internal validity is the non-deterministic nature of Large Language Models (LLMs) themselves. Models, such as Llama 3, generate outputs that can fluctuate based on parameters including input phrasing, the model's internal state, and external environmental conditions (e.g., hardware and library versions). This variability poses a problem for reproducibility as small, often barely noticeable differences can produce different results. To overcome this challenge, we made sure that there was consistency in hardware and software conditions and performed the prompting on the same machine and settings to minimize this threat.

Furthermore, manually annotated datasets add an additional layer of complications. These datasets may include inherent biases from the annotators, including subjective interpretations of code smells. Such biases may affect the accuracy and correctness of the results, complicating the determination of whether the outcomes represent the code's quality or are superficial due to the datasets' nature. In order to mitigate this threat, we used established databases that were employed in prior research instead of developing a new one and contributing to the existing bias.

It is also worth mentioning that the mapping between code smells and rules from static analyzers is not totally and universally agreed upon. Inconsistencies or incomplete mappings can lead to gaps in detecting certain code smells, potentially skewing the analysis and making comparisons across different tools unreliable. In order to reduce this risk, we searched for the existing rules in the documentation of used analyzers trying to find correspondences among them [3, 37, 50] so that the approach is more comprehensive and coherent.

5.2 External Validity

External validity concerns the applicability of our conclusions beyond the particular context and scope of this thesis. A significant constraint is our choice to confine the research to five code smells within Java code. This emphasis guarantees depth within a limited domain, but

the results may not be applicable to other programming languages. Differences in syntax, semantics, and idiomatic conventions among languages may restrict the relevance of our conclusions to circumstances outside of Java and the five selected code smells.

Furthermore, the particular attributes of the datasets and the dependence on Java-centric code smells may impede the wider applicability of the research. Other programming languages may display distinct code smells or emphasize varying facets of code quality, hence diminishing the relevance of our findings in those instances.

The issue of generalizability also extends to the use of specific LLM. Despite the fact that the research in this study is based on Llama 3, it is not to be expected that the findings of this study can be directly applied to other LLMs. Other models may perform different fine-tuning or tokenization which may lead to different results. This restricts the generalizability of our findings and therefore calls for more research across a number of models to establish these differences.

Similarly, the relevance of our findings may not be precise for future generations of LLMs. It is highly likely that the next generations will incorporate improvements in the architecture, training, or perform domain-specific enhancement. Our study thus tries to capture the current trends while preserving resource efficiency and feasibility. However, we cannot presume that it remains relevant for future versions or the fast-evolving LLMs.

Another possibility is to fine-tune a specific LLM for the task of code smell detection. Even though general-purpose models like Llama 3 or ChatGPT are highly versatile and flexible, a model that has been specifically trained on code smells datasets might deliver better results. In this study, our primary goal was to evaluate the out-of-the-box performance of a general-purpose model, with no additional fine-tuning, to understand their baseline capabilities in a realistic scenario.

Therefore, it is also important to note that the study has focused on specific code smells that are evident in Java language. A LLM might behave differently when identifying other code smells especially those that are based on different software design principles or are at a higher level of abstraction. The accuracy of LLMs in identifying such smells is yet to be established because they may complement the model's perception of the world more or contradict it due to having different experiences and training sets. This limitation highlights the need for more research that involves more types of code smells across a number of programming languages and levels of abstraction for a better understanding.

Finally, it can also be argued that not all the findings from academic research can be applied in industries directly. The experimental design, data collected, and analysis techniques employed might not be able to capture all the aspects and issues that could lead to biased results and make the findings non-transferable. Thus, keeping in mind the above, we purposely selected the datasets that contain code samples from real-industrial software projects, hence augmenting the relevance and applicability of our findings to practical software development settings.

6 Concluding Remarks

6.1 Conclusion

The process of identifying and prioritizing code smells is crucial for maintaining software quality and managing overall technical debt. Having reliable, energy-efficient, precise, and fast methods can spare a significant amount of developers' effort. Modern code smell analysis approaches are not ideal and fail to fulfill at least one of the criteria described above. To address this issue, our study aimed at assessing the effectiveness of Llama 3, a state-of-the-art LLM, in this matter.

We concluded that Llama 3 performs well in detecting structural smells like *Data Class*, aligning with related research on machine learning models. However, its performance is inconsistent for nuanced, context-dependent smells like *Feature envy* or *Multifaceted abstraction*. Although the tool performed better than Checkstyle and PMD in certain aspects, it was outperformed by SonarQube in most cases and a statistical analysis revealed that there is no significant difference between the overall performance of Llama 3 compared to other static analyzers, i.e, PMD, Checkstyle, and SonarQube. In addition, the capability of Llama 3 in prioritizing code smells as shown by its MRR score, shows the potential of the model while at the same time pointing to its weaknesses—the model often does not identify the most critical issue as the most prominent. Considering all of the above, we conclude that while not being able to substitute modern code smell analysis techniques entirely, LLMs might prove usefull as complementary tools in improving software quality.

Furthermore, some of questions were not addressed in our research and remain open. For example, how do training data influence the performance of LLMs? Would fine-tuning enhance their performance for complex smells? Referring to our conclusion, what hybrid approaches could effectively integrate LLMs with static analyzers to leverage their respective strengths, is also an open question.

6.2 Future Work

As discussed in the previous section, this research has revealed the strengths and weaknesses of Llama 3 in identifying and ranking code smells which presents new research directions. Our research can become a solid foundation for further investigation on how the fine-tuning could influence the performance of the model. By choosing another dataset which contains more labeled samples of complex code smells, emphasizing their contextual nuances, we could achieve better results. Furthermore, datasets that contain paired entries of smelly

code snippets together with their refactored versions could significantly contribute to the ability of model to learn and generalize. Hybrid approaches can involve integration of the best features of LLMs with static analysis tools. For example, static analyzers can be used for basic smells detection by applying various metrics while LLMs can be utilized for complex smells that need deeper semantic analysis. Future experiments may include scenarios in which Llama 3 assists static analyzers to test the effectiveness of the combination in various contexts. Additionally, the same methodology can be applied to larger datasets considering another programming languages or LLMs. By addressing these open questions, future research could uncover the full potential of LLMs in the field of code smell analysis.

A

Appendix

Table A.1: Detection rules and keywords for code smell identification

Code Smell	Rules
Multifaceted Abstraction	**PMD:** God Class, TooManyMethods **Checkstyle:** ClassFanOutComplexit, MethodCount **SonarQube:** S1448 (Too many methods), S6539 (Class should not depend on an excessive number of other classes) **Llama 3:** God Class, God Object, Multifaceted Abstraction
Data Class	**PMD:** DataClass **Checkstyle:** VisibilityModifier, NoFinalizer **SonarQube:** S1170 (Public constants should be static), S1104 (Fields should be private), S1820 (Classes should not have too many fields) **Llama 3:** Data Class
Long Parameter List	**PMD:** ExcessiveParameterList, ConstructorWithTooManyParameters **Checkstyle:** ParameterNumber **SonarQube:** S107 (Methods should not have too many parameters) **Llama 3:** Long Parameter List
Feature Envy	**PMD:** LawOfDemeter, CouplingBetweenObjects, ExcessiveImports **Checkstyle:** ClassDataAbstractionCoupling **SonarQube:** S1200 (Classes should not be coupled to too many other classes) **Llama 3:** Feature Envy
Complex Method	**PMD:** CyclomaticComplexity, NPathComplexity, ExcessiveMethodLength **Checkstyle:** CyclomaticComplexity, MethodLength, ExecutableStatementCount **SonarQube:** S3776 (Cognitive Complexity), S1541 (Method complexity) , S6541 (Brain method) **Llama 3:** Complex Method, Long Method

Table A.2: Overview of the detailed results per code smell across all analyzers. TP: True Positives, FP: False Positives, FN: False Negatives, TN: True Negatives

Analyzer	Code Smell	Metrics						
		TP	FP	FN	TN	Precision	Recall	F1-Score
PMD	Data Class	393	32	3134	3034	0.9245	0.1113	0.1987
	Feature Envy	633	1449	2433	2078	0.304	0.2064	0.2459
	Multifaceted Abstraction	158	398	1884	9322	0.2834	0.0772	0.1213
	Long Parameter List	365	10	864	10522	0.9725	0.2966	0.4546
	Complex Method	818	117	399	10428	0.8754	0.6723	0.7605
	None	4065	1962	3182	2552	0.6745	0.5609	0.6125
Llama 3	Data Class	3190	1155	337	1911	0.7342	0.9045	**0.8105**
	Feature Envy	239	1729	2827	1798	0.1215	0.0779	0.095
	Multifaceted Abstraction	232	387	1809	9333	0.3745	0.1135	0.1742
	Long Parameter List	507	203	722	10329	0.7142	0.4127	0.5231
	Complex Method	744	212	473	10332	0.7783	0.6112	0.6847
	None	3162	591	4085	3923	0.8425	0.4363	0.5749
Checkstyle	Data Class	17	176	3510	2890	0.087	0.0048	0.009
	Feature Envy	324	2983	2742	545	0.098	0.1057	0.1017
	Multifaceted Abstraction	413	492	1629	9228	0.4561	0.2021	0.2801
	Long Parameter List	472	21	757	10511	0.9576	0.3837	0.5479
	Complex Method	806	113	411	10431	0.8767	0.6622	0.7545
	None	4508	1532	2739	2982	0.7463	0.6219	0.6785
SonarQube	Data Class	3418	2285	109	782	0.5994	0.9692	0.7407
	Feature Envy	2227	2239	839	1288	0.4987	0.7264	0.5914
	Multifaceted Abstraction	1821	1901	220	7819	0.4892	0.8921	0.6319
	Long Parameter List	369	63	860	10469	0.8547	0.3005	0.4447
	Complex Method	715	288	502	10256	0.7128	0.5878	0.6443
	None	934	233	6313	4281	0.8004	0.1289	0.2221

Statement on the Usage of Generative Digital Assistants

For this thesis, the following generative digital assistants have been used:

We have used ChatGPT-4o (OpenAI, version December 2024) [33] for collecting possible inspiration and exploring alternative phrasing or structuring ideas in certain parts of the thesis. We are aware of the potential dangers of using these tools and have used them sensibly with caution and with critical thinking.

In particular, it was utilized to brainstorm different ways of visual representation of figures in Chapter 2 and Chapter 4, get inspiration for the phrasing of the text in Chapter 6. The above mentioned tool was used solely as a source of inspiration, and the final text was created using own words and phrasing.

Bibliography

[1] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis." In: *Information and Software Technology* 108 (2019), pp. 115–138. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.12.009.

[2] Oliver Bodemer. "Revolutionizing Software Development: Harnessing the Power of Artificial Intelligence for Next-Generation Coding Solutions." In: (Nov. 2023). DOI: 10.36227/techrxiv.24592335.v1. URL: http://dx.doi.org/10.36227/techrxiv.24592335.v1.

[3] *Checkstyle*. Website. Available online at https://checkstyle.sourceforge.io/.

[4] Celia Chen, Reem Alfayez, Kamonphop Srisopha, Barry Boehm, and Lin Shi. "Why Is It Important to Measure Maintainability and What Are the Best Ways to Do It?" In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 377–378.

[5] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374.

[6] Abhimanyu Dubey et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783.

[7] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. "Comparing and experimenting machine learning techniques for code smell detection." In: *Empirical Software Engineering* 21.3 (2016), pp. 1143–1191. ISSN: 1573-7616. DOI: 10.1007/s10664-015-9378-4. URL: https://doi.org/10.1007/s10664-015-9378-4.

[8] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201485672.

[9] GitHub. *How Generative AI is Changing the Way Developers Work*. 2023. URL: https://github.blog/ai-and-ml/generative-ai/how-generative-ai-is-changing-the-way-developers-work/.

[10] GitHub. *GitHub Copilot: Your AI Pair Programmer*. 2024. URL: https://github.com/features/copilot.

[11] Google AI. *PaLM 2: The Next Generation Large Language Model*. 2023. URL: https://ai.google/discover/palm2/.

[12] Mouna Hadj-Kacem and Nadia Bouassida. "Application of Deep Learning for Code Smell Detection: Challenges and Opportunities." In: *SN Computer Science* 5.5 (2024), p. 614. ISSN: 2661-8907. DOI: 10.1007/s42979-024-02956-5. URL: https://doi.org/10.1007/s42979-024-02956-5.

[13] Desta Haileselassie Hagos, Rick Battle, and Danda B. Rawat. *Recent Advances in Generative AI and Large Language Models: Current Status, Challenges, and Perspectives*. 2024. arXiv: 2407.14962 [cs.CL]. URL: https://arxiv.org/abs/2407.14962.

[14] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. "A Systematic Review of Fault Prediction Performance in Software Engineering." In: *Software Engineering, IEEE Transactions on* 38 (Jan. 2011), pp. 1 –1. DOI: 10.1109/TSE.2011.103.

[15] Xiaofeng Han, Amjed Tahir, Peng Liang, Steve Counsell, and Yajing Luo. "Understanding Code Smell Detection via Code Review: A Study of the OpenStack Community." In: Mar. 2021. DOI: 10.1109/ICPC52881.2021.00038.

[16] Anh Ho, Anh Bui, Phuong Nguyen, and Amleto Di Salle. "Fusion of deep convolutional and LSTM recurrent neural networks for automated detection of code smells." In: June 2023, pp. 229–234. DOI: 10.1145/3593434.3593476.

[17] Arvinder Kaur and Ruchikaa Nayyar. "A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code." In: *Procedia Computer Science* 171 (Jan. 2020), pp. 2023–2029. DOI: 10.1016/j.procs.2020.04.217.

[18] Peter Kokol, Marko Kokol, and Sašo Zagoranski. *Code smells: A Synthetic Narrative Review*. 2021. arXiv: 2103.01088 [cs.SE]. URL: https://arxiv.org/abs/2103.01088.

[19] Peter Kokol, Marko Kokol, and Sašo Zagoranski. *Code smells: A Synthetic Narrative Review*. 2021. arXiv: 2103.01088 [cs.SE]. URL: https://arxiv.org/abs/2103.01088.

[20] William H. Kruskal and W. Allen Wallis. "Use of Ranks in One-Criterion Variance Analysis." In: *Journal of the American Statistical Association* 47.260 (1952), pp. 583–621. ISSN: 01621459, 1537274X. URL: http://www.jstor.org/stable/2280779 (visited on 12/06/2024).

[21] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D. Le, and David Lo. *Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues*. 2023. arXiv: 2307.12596 [cs.SE]. URL: https://arxiv.org/abs/2307.12596.

[22] Naseef M.N.M, Nifthas A.R.M, and * KapilanK. "Analyzing the Impact of Code Smells On Software Maintainability." In: (May 2023). DOI: 10.22541/au.168552004.43700626/v1. URL: http://dx.doi.org/10.22541/au.168552004.43700626/v1.

[23] Lech Madeyski and Tomasz Lewowski. "MLCQ: Industry-Relevant Code Smell Data Set." In: Apr. 2020, pp. 342–347. DOI: 10.1145/3383219.3383264.

[24] R. Marinescu. "Detection strategies: metrics-based rules for detecting design flaws." In: *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* 2004, pp. 350–359. DOI: 10.1109/ICSM.2004.1357820.

[25] Robert Martin. *Agile Software Development, Principles, Patterns, and Practices*. Pearson Deutschland, 2013, pp. 95–99. ISBN: 9781292025940. URL: https://elibrary.pearson.de/book/99.150005/9781292038360.

[26] Steve McConnell. *Code Complete, Second Edition*. USA: Microsoft Press, 2004. ISBN: 0735619670.

[27] John H. McDonald. *Handbook of Biological Statistics*. 3rd. Baltimore, Maryland: Sparky House Publishing, 2014, pp. 157–165.

[28] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. "DECOR: A Method for the Specification and Detection of Code and Design Smells." In: *IEEE Transactions on Software Engineering* 36.1 (2010), pp. 20–36. DOI: 10.1109/TSE.2009.50.

[29] María Isabel Murillo and Marcelo Jenkins. "Technical Debt Measurement during Software Development using Sonarqube: Literature Review and a Case Study." In: *2021 IEEE V Jornadas Costarricenses de Investigación en Computación e Informática (JoCICI)*. 2021, pp. 1–6. DOI: 10.1109/JoCICI54528.2021.9794341.

[30] Mika V. Mäntylä and Casper Lassenius. "Subjective evaluation of software evolvability using code smells: An empirical study." In: *Empirical Software Engineering* 11.3 (2006), pp. 395–431. ISSN: 1573-7616. DOI: 10.1007/s10664-006-9002-8. URL: https://doi.org/10.1007/s10664-006-9002-8.

[31] Himesh Nandani, Mootez Saad, and Tushar Sharma. *DACOS-A Manually Annotated Dataset of Code Smells*. 2023. arXiv: 2303.08729 [cs.SE]. URL: https://arxiv.org/abs/2303.08729.

[32] Allen Hsu Somakala Jagannathan Sajjad Mustehsan Session Mwamufiya Marc Novakouski. "Analysis Tool Evaluation: PMD." In: Apr. 2007.

[33] OpenAI. *ChatGPT: Language Model*. Website. Available online at :https://openai.com/index/chatgpt/.

[34] OpenAI. *DALL-E 2: AI System for Creating Images from Text*. 2024. URL: https://openai.com/index/dall-e-2/.

[35] Oracle. *DataTruncation Class Documentation*. Accessed: 2024-09-04. 2023. URL: https://docs.oracle.com/en/java/javase/17/docs/api/java.sql/java/sql/DataTruncation.html.

[36] Stack Overflow. *Stack Overflow Annual Developer Survey*. Website. Available online at https://survey.stackoverflow.co/2023/#developer-profile-demographics. 2023.

[37] *PMD*. Website. Available online at https://docs.pmd-code.org/latest/.

[38] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. "On the evaluation of code smells and detection tools." In: *Journal of Software Engineering Research and Development* 5.1 (2017), p. 7. ISSN: 2195-1721. DOI: 10.1186/s40411-017-0041-1. URL: https://doi.org/10.1186/s40411-017-0041-1.

[39] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation." In: *Empirical Software Engineering* 23.3 (2018), pp. 1188–1221. ISSN: 1573-7616. DOI: 10.1007/s10664-017-9535-z. URL: https://doi.org/10.1007/s10664-017-9535-z.

[40] Md. Masudur Rahman, Abdus Satter, Md. Mahbubul Alam Joarder, and Kazi Sakib. "An Empirical Study on the Occurrences of Code Smells in Open Source and Industrial Projects." In: *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '22. Helsinki, Finland: Association for Computing Machinery, 2022, 289–294. ISBN: 9781450394277. DOI: 10.1145/3544902.3546634. URL: https://doi.org/10.1145/3544902.3546634.

[41] Zeeshan Rasheed, Malik Abdul Sami, Muhammad Waseem, Kai-Kristian Kemell, Xiaofeng Wang, Anh Nguyen, Kari Systä, and Pekka Abrahamsson. *AI-powered Code Review with LLMs: Early Results*. 2024. arXiv: 2404.18496 [cs.SE]. URL: https://arxiv.org/abs/2404.18496.

[42] Denise Rey and Markus Neuhäuser. "Wilcoxon-Signed-Rank Test." In: *International Encyclopedia of Statistical Science*. Ed. by Miodrag Lovric. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1658–1659. ISBN: 978-3-642-04898-2. DOI: 10.1007/978-3-642-04898-2_616. URL: https://doi.org/10.1007/978-3-642-04898-2_616.

[43] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. *A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications*. 2024. arXiv: 2402.07927 [cs.AI]. URL: https://arxiv.org/abs/2402.07927.

[44] Kathrin Seßler, Maurice Fürstenberg, Babette Bühler, and Enkelejda Kasneci. *Can AI grade your essays? A comparative analysis of large language models and teacher ratings in multidimensional essay scoring*. 2024. arXiv: 2411.16337 [cs.CL]. URL: https://arxiv.org/abs/2411.16337.

[45] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. "Code smell detection by deep direct-learning and transfer-learning." In: *Journal of Systems and Software* 176 (June 2021), p. 110936. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.110936. URL: http://dx.doi.org/10.1016/j.jss.2021.110936.

[46] Luciana Silva, Janio Silva, João Montandon, and Marco Valente. *Detecting Code Smells using ChatGPT: Initial Insights*. July 2024. DOI: 10.13140/RG.2.2.36634.04802.

[47] Luciana Silva, Janio Silva, João Montandon, and Marco Valente. *Detecting Code Smells using ChatGPT: Initial Insights*. July 2024. DOI: 10.13140/RG.2.2.36634.04802.

[48] Elder Sobrinho and Marcelo Maia. "On the Interplay of Smells Large Class, Complex Class and Duplicate Code." In: *Proceedings of the XXXV Brazilian Symposium on Software Engineering*. SBES '21. Joinville, Brazil: Association for Computing Machinery, 2021, 64–73. ISBN: 9781450390613. DOI: 10.1145/3474624.3474716. URL: https://doi.org/10.1145/3474624.3474716.

[49] Zéphyrin Soh, Foutse Khomh, and Yann-Gaël Guéhéneuc. "Do Code Smells Impact the Effort of Different Maintenance Programming Activities?" In: Mar. 2016, pp. 393–402. DOI: 10.1109/SANER.2016.103.

[50] *SonarQube*. Website. Available online at https://www.sonarsource.com/.

[51] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: https://arxiv.org/abs/2302.13971.

[52] Di Wu, Fangwen Mu, Lin Shi, Zhaoqiang Guo, Kui Liu, Weiguang Zhuang, Yuqi Zhong, and Li Zhang. "iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring." In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24. Sacramento, CA, USA: Association for Computing Machinery, 2024, 1345–1357. ISBN: 9798400712487. DOI: 10.1145/3691620.3695508. URL: https://doi.org/10.1145/3691620.3695508.

[53] Jones Yeboah and Saheed Popoola. "Efficacy of Static Analysis Tools for Software Defect Detection on Open-Source Projects." In: *2023 International Conference on Computational Science and Computational Intelligence (CSCI)*. 2023, pp. 1588–1593. DOI: 10.1109/CSCI62032.2023.00262.

[54] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. *Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT*. 2023. arXiv: 2304.10778 [cs.SE]. URL: https://arxiv.org/abs/2304.10778.