UNIVERSITY OF CALIFORNIA,
IRVINE

Exploring the Efficacy of of GPT-3.5 in Code Smell Detection

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Yang Liu

2024

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my thesis advisor and committee chair, Professor Iftekhar Ahmed, for being my mentor and supporting my research journey throughout my entire Master's program. His patience and encouragement gave me confidence, and his insights shaped my wonderful research experience. Without his guidance and persistent help, this thesis would not have been possible.

I would also like to thank my committee members, Professor James A. Jones, Professor Joshua Garcia, and Professor Mohammad Moshirpour, for their insightful questions, advice, and encouragement.

Finally, I would like to thank the ICS Graduate Office for their guidance with the paperwork for my graduation.

ABSTRACT OF THE THESIS

Exploring the Efficacy of of GPT-3.5 in Code Smell Detection

By

Yang Liu

Master of Science in Software Engineering

University of California, Irvine, 2024

Assistant Professor Iftekhar Ahmed, Chair

Code smell is a widely recognized term in the software community, used to describe low-quality software designs. It adversely affects the comprehensibility and maintainability of software systems, ultimately reducing their overall quality. Despite the development of various static analysis tools for identifying code smells, many complex code smells are not properly detected because they cannot be fully captured by specific rules and patterns. The emergence of Large Language Models (LLMs) provides a new opportunity to identify code smells through natural language processing, which may be advantageous for detecting complex code smells. In this paper, we present experiments exploring the efficacy of *GPT-3.5* in code smell detection, conducted on 14 Open-Source Software (OSS) projects, totaling 9,740 Python files. We select three code smells in Python to assess: too many parameters (TMP), too many nested blocks (TMNB), and unused variable (UV). We use the static analysis tool *Pylint* to automate the detection of code smells and create a dataset. Subsequently, we fine-tune the *GPT-3.5* model specifically for code smell detection. To evaluate the performance of the fine-tuned model, we apply it to four additional OSS projects. Our results demonstrate that *GPT-3.5* has the potential to replace traditional static analysis tools in code smell detection. However, it still requires more data and refinement to improve accuracy and reliability for certain smells. Additionally, we reveal that the size of the dataset significantly influences the performance of *GPT-3.5*. We also present conjectures for future research in

code smell detection and LLM fine-tuning.

Chapter 1

Introduction

1.1 Background

While code smells may not instantly affect the code's functionality, they can eventually result in maintenance difficulties, diminished readability, and a greater risk of introducing bugs[6]. Code smells function as early indicators of potential design flaws or bugs within the software system.

Static analysis tools have been developed to automatically detect code smells and other issues in software systems. These tools analyze the source code without executing it. Early efforts primarily revolved around heuristic-based methods. Relying on predefined rules and patterns, these approaches identified code smells such as God Class, Feature Envy, and Duplicated Code by analyzing structural and semantic characteristics of source code[6]. Over the years, static analysis tools have evolved to become more sophisticated, employing various algorithms and heuristics to detect a wide range of code smells. Many formulating metrics-based rules that capture deviations from good design principles and heuristics are applied in identifying design flaw[13]. These approaches leverage features extracted from source code, including metrics like cyclomatic complexity, code churn, and code coupling, to automatically identify potential code smells. Popular static analysis tools include Pylint [12], PMD [17], and Checkstyle [3].

Static analysis tools have been developed to automatically detect code smells and other issues in software systems. These tools analyze source code without executing it. Early efforts primarily revolved around heuristic-based methods, relying on predefined rules and patterns to identify code smells such as God Class, Feature Envy, and Duplicated Code by analyzing structural and semantic characteristics of the source code[6]. Over the years, static analysis tools have evolved to become more sophisticated, employing various algorithms and heuristics to detect a wide range of code smells. Many of these tools formulate metrics-based rules that capture deviations from good design principles, applying heuristics to identify design flaws[13]. These approaches leverage features extracted from source code, including metrics like cyclomatic complexity, code churn, and code coupling, to automatically identify potential code smells. Popular static analysis tools include Pylint[12], PMD[17], and Checkstyle[3].

While these tools offer significant advantages in maintaining code quality, they also come with certain disadvantages. Code smell detectors often produce varying results and rarely agree with each other [5]. Typically, these tools operate by calculating a set of metrics to identify specific smells. The metrics used can vary; for instance, in detecting the Large Class smell, one tool might employ diverse cohesion and complexity metrics, while another might rely solely on the lines of code (LOC) metric. Furthermore, even when the same metrics are used, the threshold values for these metrics can be different. Another problem regards the high configuration effort. Setting up static analysis tools to align with the specific coding standards and requirements of a software can be time-consuming, and previous findings indicate that developers are often unwilling to configure these tools [19].

In recent years, the application of machine learning (ML) techniques in code smell detection has garnered considerable attention from the software engineering community. Palomba et al. [16] proposed an approach to identify five distinct code smells—namely, divergent change, shotgun surgery, parallel inheritance, blob, and feature envy—by leveraging change history information mined from version control systems. Gupta et al. [10] introduced a novel method for detecting code smells in heterogeneous data using a modified domain invariant transfer kernel learning (DITKL), which is a technique in transfer learning. Das et al. [4] presented a deep learning-based approach to detect two specific code smells: Brain Class and Brain Method.

OpenAI's GPT-3.5, a state-of-art NLP model, has showcased exceptional abilities in comprehending and generating human-like text [2]. Trained on an extensive dataset comprising billions of parameters, GPT-3.5 is capable of assimilating a vast repository of linguistic knowledge. This comprehensive pretraining endows it with the ability to contextually understand code. Moreover, GPT-3.5 can be fine-tuned on domain-specific datasets or adapted to specific tasks through further training, facilitating continual learning and adaptation to evolving requirements [11]. This adaptability enables the model to be customized for code smell detection, significantly reducing the time required for tool configuration.

GPT-3.5 has the potential to overcome the limitations of traditional static analysis tools in detecting code smells. Our research aims to explore the efficacy of GPT-3.5 for this purpose. We focused on three specific code smells in Python: too many parameters, too many nested blocks, and unused variable. We begin by using the static analysis tool Pylint to identify these code smells across 14 Open-Source Software (OSS) projects, comprising a total of 9,740 Python files sourced from GitHub. This process enables the creation of a comprehensive dataset. Subsequently, we fine-tune a new GPT-3.5 model specifically for detecting these code smells. To evaluate the performance of our model, we applied it to four additional OSS projects.

1.2 Research Questions

Specifically, this paper addresses the following research questions:

- RQ1: How effective is GPT-3.5 in detecting code smells?
- RQ2: Which code smells does GPT-3.5 effectively detect?
- RQ3: What impact does the size of dataset have on the performance of GPT-3.5 in code smell detection?

1.3 Contributions and Organization of the Thesis

The contributions of this paper are listed below:

- We explore the potential of using large language models in identifying code smells.
- We report the results on the effectiveness of GPT-3.5 in detecting code smells.
- We explore the factors that might impact the performance of GPT-3.5 in identifying code smells.
- We propose several conjectures for future research on code smell detection and the fine-tuning of large language models.

The remainder of the paper is organized as follows. Chapter 2 introduces the methodology in detail, smelly code extraction, dataset creation, and fine-tuning. Chapter 3 presents our results and findings. Chapter 4 discuss some conjectures inspired from our results. Chapter 5 concludes with a summary of the key findings and future work.

Chapter 2

Methodology

This study aims to investigate the potential of GPT-3.5 in detecting code smells. Specifically, we focus on identifying code smells in Python, one of the most widely used programming languages globally. GPT-3.5, with its extensive access to Python-related data, is well-suited for this task. To evaluate the performance of GPT-3.5, we employ Pylint, a source code analyzer that identifies programming errors and enforces coding standards.

We examine three code smells: too many parameters, too many nested blocks, and unused variable. Initially, we select 14 open-source software projects, totaling 9,740 Python files, primarily written in Python and sourced from GitHub. These projects are chosen based on their number of stars on GitHub. We then use Pylint to automatically detect the three code smells in these software projects, thus creating a dataset for GPT-3.5. Subsequently, we fine-tune a new GPT-3.5 model. Finally, we apply the newly fine-tuned model to four additional software projects from GitHub and evaluated its performance. Furthermore, we investigate the impact of the dataset on GPT-3.5's performance. Figure 2.1 presents the overall procedural steps of our methodology. The subsequent sections will provide a comprehensive explanation of each phase.

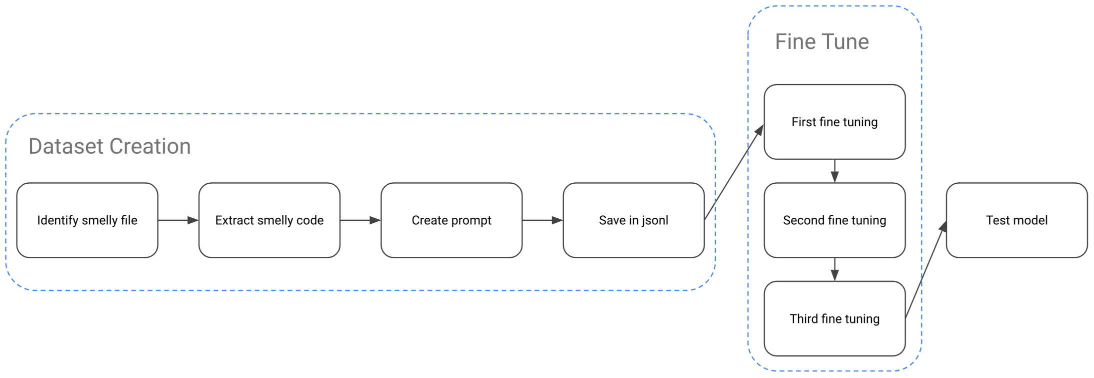


Figure 2.1: Methodology procedures

2.1 Code Smell Selection and Description

In the investigation conducted by prior research [8], a list of five code smells was identified, which violate the design structure of code and hinder its maintainability and readability based on their severity, as shown in Table 2.1. In the severity column, numerical values denote the intensity levels of these code smells. A rating within the range of [5.5-7.75) is classified as moderate, while those within [7.75-10) are classified as major. Due to the constraints imposed by Pylint, our study focuses on three specific code smells that it can detect: ‘Collapsible IF’, ‘Many Parameter’, and ‘Unused Variable’.

Table 2.1: Severity index of code smells [8]

Python Code Smells	Severity
Cognitive Complexity	8.01
Collapsible ‘IF’	7.37
Many Parameter	7.02
Naming Conventions	6.22
Unused Variable	7.07

In Pylint’s terminology, ‘Collapsible IF’ is described as ‘too many nested blocks’, while ‘Many Parameter’ is termed ‘too many parameters’. Our research aligns with Pylint’s nomenclature for these code smells.

2.1.1 Too Many Parameters (TMP)

'Too many parameters' can arise when an excessive number of inputs are provided to a single method [9]. In an effort to generalize a routine with multiple variations, a developer may inadvertently pass too many parameters. Another possible cause is a lack of understanding of the relationships between objects, leading to the inclusion of all entities as parameters instead [20].

Essentially, the longer the parameter list, the harder it is to comprehend. We align with Pylint's default setting, which limits the maximum number of parameters to five. Figure 2.2 presents an illustrative example. The code snippet on the left features a smelly function with 13 parameters, far exceeding the limit of five. Conversely, the non-smelly code snippet on the right adheres to the limit, containing no more than five parameters.

```python
def three_d_chess_move(
    x_white,
    y_white,
    z_white,
    piece_white,
    x_black,
    y_black,
    z_black,
    piece_black,
    x_blue,
    y_blue,
    z_blue,
    piece_blue,
    current_player,
):
    pass


```

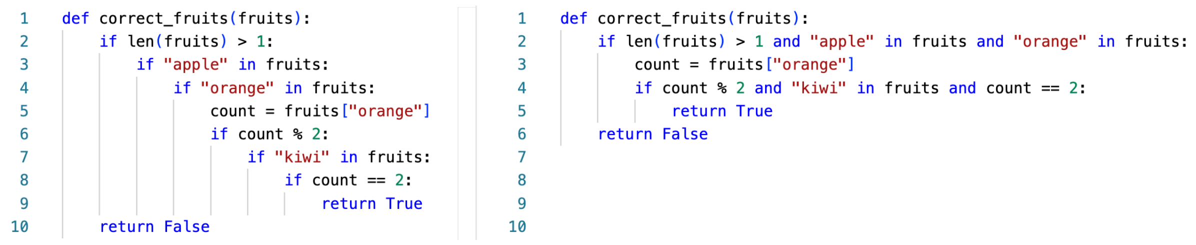
```python
from dataclasses import dataclass


@dataclass
class ThreeDChessPiece:
    x: int
    y: int
    z: int
    type: str


def three_d_chess_move(
    white: ThreeDChessPiece,
    black: ThreeDChessPiece,
    blue: ThreeDChessPiece,
    current_player,
):
    pass
```

Figure 2.2: Smelly code (too many parameters, left) and non-smelly code (right)

2.1.2 Too Many Nested Blocks (TMNB)

'Too many nested blocks' refers to a coding practice where multiple nested statements can be consolidated into a single statement with compound conditions [20]. This structure tends to become increasingly complex over time as developers continue to add conditions and additional levels of nesting [21].

Initially, we use Pylint's default setting, which limits the maximum number of nested blocks to five. However, we find this threshold to be unreasonable and subsequently adjust it to three. The detailed reasoning behind this adjustment is discussed in Chapter 2. Figure 2.3 illustrates the concept: the smelly code on the left contains six nested blocks, exceeding the threshold of three, whereas the non-smelly code on the right contains only two nested blocks.



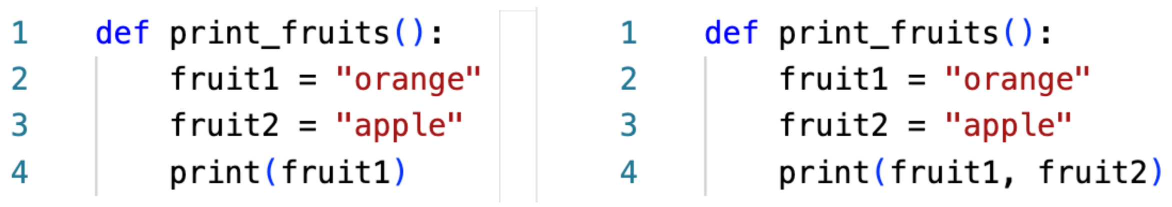
```python
def correct_fruits(fruits):
    if len(fruits) > 1:
        if "apple" in fruits:
            if "orange" in fruits:
                count = fruits["orange"]
                if count % 2:
                    if "kiwi" in fruits:
                        if count == 2:
                            return True
    return False
```

```python
def correct_fruits(fruits):
    if len(fruits) > 1 and "apple" in fruits and "orange" in fruits:
        count = fruits["orange"]
        if count % 2 and "kiwi" in fruits and count == 2:
            return True
    return False
```

Figure 2.3: Smelly code (too many nested blocks, left) and non-smelly code (right)

2.1.3 Unused Variable (UV)

'Unused variable' refers to a variable that has been declared but is not used or referenced elsewhere in the code [8]. Such variables can lead to confusion, misunderstandings, and hinder code maintainability. In Figure 2.4, the smelly code on the left contains an unused variable, 'fruit2'.



```python
def print_fruits():
    fruit1 = "orange"
    fruit2 = "apple"
    print(fruit1)
```

```python
def print_fruits():
    fruit1 = "orange"
    fruit2 = "apple"
    print(fruit1, fruit2)
```

Figure 2.4: Smelly code (unused variable, left) and non-smelly code (right)

2.2 Dataset Creation

In this section, we detail the steps of selecting data sources, identifying and extracting smelly code, creating prompts for fine-tuning, formatting the data, and fine-tuning the GPT-3.5 model.

2.2.1 Software Selection

We selected 14 open-source software projects from GitHub, totaling 9,740 Python files, based on their star ratings to create our dataset. Additionally, we chose four open-source software projects, totaling 749 Python files, for testing purposes. Table 2.2 shows the star number of the software used for training, and Table 2.3 shows the star number of the software used for testing.

2.2.2 Smelly Code Detection and Extraction

We use Pylint to identify smelly Python files. Then, we extract the smelly functions to form our prompt, which is used for fine-tuning GPT-3.5, based the location information of smelly code from Pylint. Initially, we employ the thresholds, which are the default settings of Pylint, as outlined in Table 2.4, and the resulting distribution of code smells is depicted in Figure 2.5. However, a notable observation emerges: the occurrence of the 'too many nested blocks' code smell is markedly lower than the other two smells (96% less). This disparity is aberrant.

Table 2.2: Star of the software for training

Software	Star (in thousands)
Django	77.1
Fastapi	71.5
Flask	66.5
Scikit-learn	58.3
Requests	51.4
Pandas	42.1
Streamlit	32.1
Matplotlib	19.4
Click	15.1
Natural Language Toolkit	13.1
Gunicorn	9.5
Sqlalchemy	8.9
Pygame	7.0
Flask-restful	6.8

Table 2.3: Star of the software for testing

Software	Star (in thousands)
Scrapy	51.2
Celery	23.6
Tornado	21.5
Boto3	8.7

Upon closer examination, the threshold set for this smell is unreasonable in practical coding practices; more than five nested blocks are infrequent in real-world scenarios. There is no consensus on an exact value in existing research.

To facilitate further research, we adjust the threshold to three, a more empirically reasonable value, enabling us to capture a broader range of relevant data. Additionally, we randomly select 791 instances of non-smell data for training. The distribution of code smells with updated thresholds is presented in Figure 2.6.

Table 2.4: Initial threshold setting for code smell detection

Code Smell	Threshold
Too Many Parameters	Maximum number = 5
Too Many Nested Blocks	Maximum number = 5
Unused Variable	None

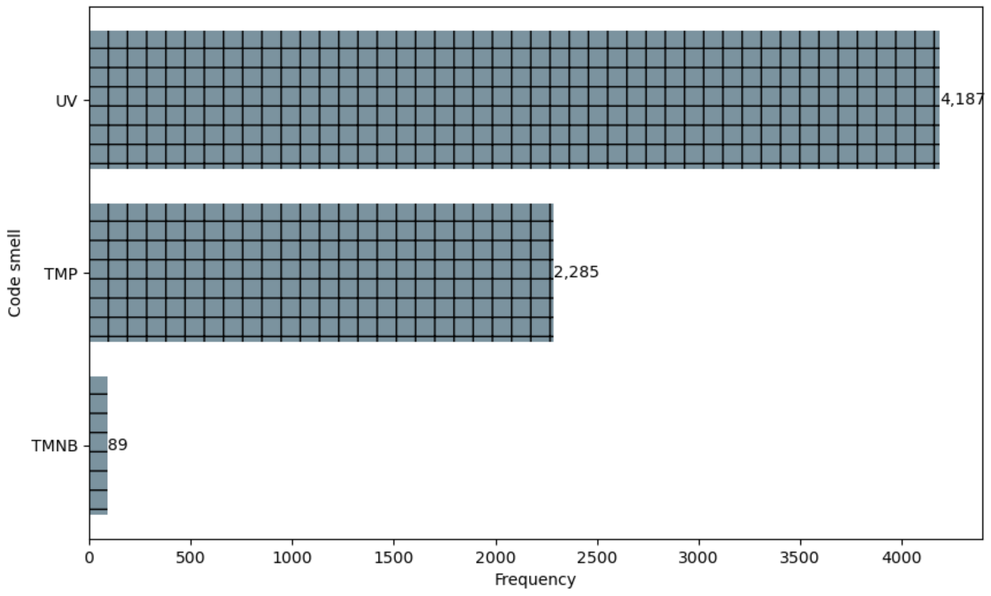


Figure 2.5: Code smell distribution (**TMP**: too many parameters, **TMNB**: too many nested blocks, **UV**: unused variable)

2.2.3 Prompt

A prompt in generative AI models refers to the textual input provided by users to guide the model's output. This input ranges from simple questions to detailed descriptions or specific tasks. In the context of GPT-3.5, prompts vary from straightforward queries to intricate problem statements [1]. Basic prompts in GPT-3.5 involve asking direct questions or providing straightforward instructions for a task. Advanced prompts, on the other hand, employ more complex structures, such as 'chain of thought' prompting, where the model is guided to follow a logical reasoning process to reach an answer. Following the guidance from prior research [7], we develop an advanced prompt with detailed descriptions to fine-tuned GPT-3.5, as shown in Table 2.5.

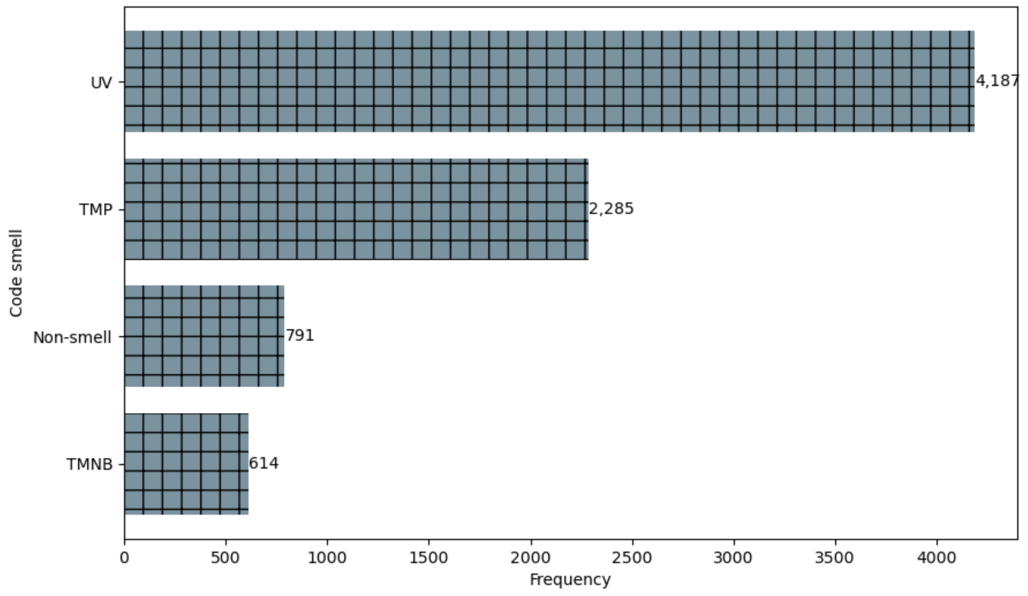


Figure 2.6: Code smell distribution with new thresholds (**TMP**: too many parameters, **TMNB**: too many nested blocks, **UV**: unused variable)

2.2.4 Data Format

As required by OpenAI, the dataset for fine-tuning must adhere to a specific format [15]. This format consists of three parts: system, user, and assistant.

System

The system part describes the model's functionality, as detailed in Table 2.6.

User

As shown in Table 2.7, the user segment simulates a scenario in which a user interacts with GPT-3.0. The advanced prompt is included in the content.

Table 2.5: Advanced prompt

1	Below are descriptions of three prevalent code smells:
2	1. Too many parameters: This smell occurs when more than five (including five) parameters are provided as input for a single method.
3	2. Too many nested blocks: This smell occurs when three or more if statements are nested together.
4	3. Unused variable: This smell occurs when a variable has been declared but it is not being used or referenced anywhere else in the code.
5	How many code smells (too many parameters, too many nested blocks, and unused variable) are present in the following Python code?
6	Python function
7	You are required to respond in the following JSON format. “Code smell” refers to the name of the smell, and “Message” describes why the smell is identified. If there are no code smells detected, return an empty list [] in JSON format.
8	[
9	{
10	“Code smell”: “Too many parameters”,
11	“Message”: “Too many arguments (6/5)”
12	},
13	{
14	“Code smell”: “Unused variable”,
15	“Message”: “Unused variable ‘lr’ ”
16	},
17	{
18	“Code smell”: “Too many nested blocks”,
19	“Message”: “Too many nested blocks (5/3)”
20	},
21	...
22	]

Assistant

In Table 2.8, the content of the assistant represents the correct answer in JSON format, which we expect GPT-3.5 to respond with.

Table 2.6: System content

```
{
        “role”: “system”,
        “content” : “I am a chatbot that can detect the three types of code smells below exists in the Python code:”
        1. Too many parameters: This smell occurs when more than five (including five) parameters are provided as input for a single method.
        2. Too many nested blocks: This smell occurs when three or more if statements are nested together.
        3. Unused variable: This smell occurs when a variable has been declared but it is not being used or referenced anywhere else in the code.
}
```

Table 2.7: User content

```
{
        “role”: “user”,
        “content” : Advanced prompt
}
```

Table 2.8: Assistant content

```
{
        “role”: “assistant”,
        “content” : Code smell detection result in JSON format
}
```

2.3 Fine-tuning and Evaluation

2.3.1 Fine-tuning

We select GPT-3.5-turbo-0125 [14] as our base model. We fine-tune the base model three times. In the first fine-tuning, we train the model to detect two code smells: ‘too many parameters’ and ‘unused variable’. In the second fine-tuning, we train the model with non-smelly data. In the third fine-tuning, we train the model to detect the code smell: ‘too many nested blocks’.

2.3.2 Evaluation

Since these three code smells are rule-based, Pylint does not make trivial mistakes. For instance, in the case of the ”too many parameters” code smell, Pylint does not flag a function that contains only two parameters as smelly. We also randomly select smelly Python files detected by Pylint and manually verify them, finding no false positives or false negatives. Therefore, Pylint can be considered a reliable baseline for evaluating our model.

To answer RQ1 (How effective is GPT-3.5 in detecting code smells?) and RQ2 (Which code smells does GPT-3.5 effectively detect?), we report the standard precision, recall, and F1-score to assess the performance of our model. We list the formulas for measuring precision, recall, and F1-score below. We also compare the performances of GPT-3.5 in both zero-shot and four-shot settings through our experiments.

- **Precision**: The proportion of correct identifications within the predicted labels for a particular category.

$$precision = \frac{TP}{TP + FP} \tag{2.1}$$

- **Recall**: The proportion of correct identifications within a particular category based on the original labels.

$$recall = \frac{TP}{TP + FN} \tag{2.2}$$

- **F1 score**: The harmonic mean of precision and recall.

$$recall = \frac{2TP}{2TP + FP + FN} \tag{2.3}$$

To answer RQ3 (What impact does the size of dataset have on the performance of GPT-3.5 in code smell detection?), we resize the dataset and fine-tune another GPT-3.5 model. We then compare this newly fine-tuned model with the original fine-tuned model and GPT-3.5.

Chapter 3

Results

3.1 RQ1: How effective is GPT-3.5 in detecting code smells?

To answer this research question, we assess the performance of three GPT-3.5 models (zero-shot, four-shot, and fine-tuned) in identifying three code smells: too many parameters, too many nested blocks, and unused variable. The evaluation metrics used are precision, recall, and F1 score, with results summarized in Tables 3.1, 3.2, and 3.3.

Precision Analysis

Precision measures the accuracy of positive predictions. Higher precision indicates fewer false positives. Below is a summary of the analysis:

- **Too many parameters:** The four-shot model achieves the highest precision (0.67), followed by the fine-tuned model (0.63) and the zero-shot model (0.47).

- **Too many nested blocks:** The fine-tuned model achieves a precision of 0.59, outperforming the zero-shot (0.54) and four-shot (0.33) models.
- **Unused variable:** The zero-shot and four-shot models both score 0.37, whereas the fine-tuned model scores the lowest at 0.20.

Table 3.1: Precision of all models

	Zero-shot	Four-shot	Fine-tuned model
Too Many Parameters	0.47	**0.67**	0.63
Too Many Nested Blocks	0.54	0.33	**0.59**
Unused Variable	**0.37**	0.37	0.20

Observation 1: GPT-3.5 tends to generate false positives in code smell detection, necessitating additional refinement.

The precision results suggest that GPT-3.5, in its current state, is prone to generating false positives in code smell detection, as evidenced by precision values all below 0.8. This issue is particularly concerning for ‘unused variable’ detection. Even after fine-tuning, the precision does not meet the desired threshold for reliable detection. Therefore, further refinements are required to enhance GPT-3.5’s precision in accurately detecting code smells.

Recall Analysis

Recall indicates GPT-3.5’s ability to capture all relevant instances. A higher recall signifies fewer false negatives. The summary of the analysis is as follows:

- **Too many parameters:** The fine-tuned model achieves perfect recall (1.00), surpassing the zero-shot (0.99) and four-shot (0.88) models.
- **Too many nested blocks:** The fine-tuned model again achieves a recall of 0.39, outperforming the zero-shot (0.27) and four-shot (0.17) models.

- **Unused variable:** The fine-tuned models consistently achieve a perfect recall of 1.00, significantly outperforming the zero-shot (0.67) and four-shot (0.57) models.

Table 3.2: Recall of all models

	Zero-shot	Four-shot	Fine-tuned model
Too Many Parameters	0.99	0.88	**1.00**
Too Many Nested Blocks	0.27	0.17	**0.39**
Unused Variable	0.67	0.57	**1.00**

The fine-tuned model demonstrates the highest recall across all code smell categories, indicating that GPT-3.5 significantly enhances its ability to identify code smells post fine-tuning, as shown in Figure 3.1.

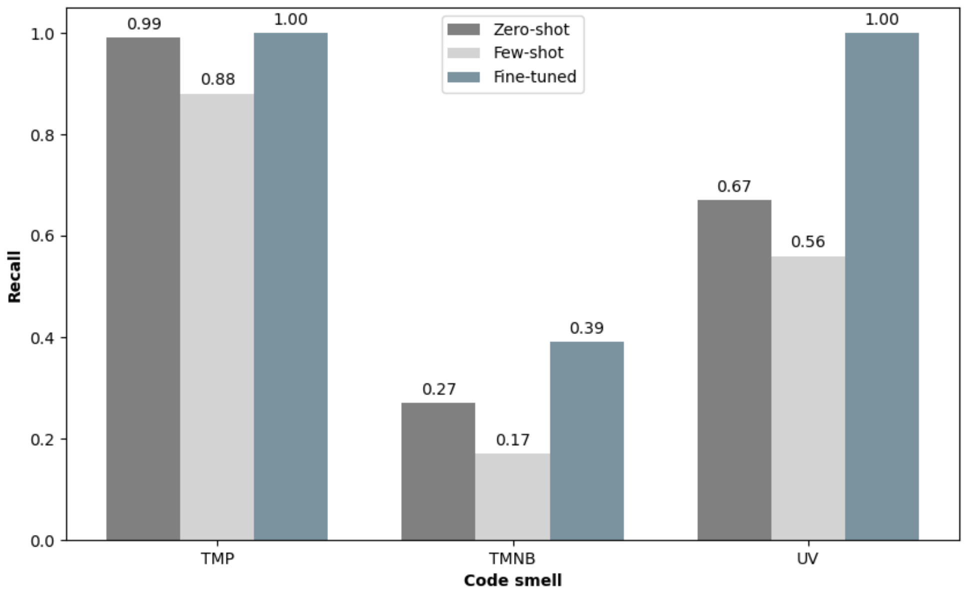


Figure 3.1: Comparison of recall (**TMP**: too many parameters, **TMNB**: too many nested blocks, **UV**: unused variable)

Observation 2: GPT-3.5 significantly enhances its ability to identify code smells after fine-tuning.

F1 Score Analysis

The F1 score, as the harmonic mean of precision and recall, serves as a balanced metric for assessing both the precision and recall of GPT-3.5's performance.

- **Too many parameters:** The fine-tuned model slightly outperforms the others with an F1 score of 0.77, followed closely by the four-shot model with a score of 0.76, and the zero-shot model with a score of 0.64.
- **Too many nested blocks:** The fine-tuned model achieves an F1 score of 0.47, outperforming the zero-shot model, which scores 0.36, and the four-shot model, which scores 0.22.
- **Unused variable:** The zero-shot model attains the highest F1 score (0.48), surpassing both the four-Shot model (0.44) and the fine-tuned model (0.34).

Table 3.3: F1 score of all models

	Zero-shot	Four-shot	Fine-tuned model
Too Many Parameters	0.64	0.76	**0.77**
Too Many Nested Blocks	0.36	0.22	**0.47**
Unused Variable	**0.48**	0.44	0.34

Observation 3: GPT-3.5 exhibits instability in code smell detection and tends to make mistakes, yet further fine-tuning may increase its reliability.

The fine-tuned model consistently demonstrates the highest F1 scores for detecting 'too many parameters' and 'too many nested blocks', while the zero-shot model excels in identifying "unused variables." However, only for 'too many parameters' do the fine-tuned models achieve an ideal F1 score of 0.77. For other code smells, all models have F1 scores lower than 0.5. These results indicate that GPT-3.5 is not consistently reliable in code smell detection and tends to make mistakes. Fine-tuning, however, improves the stability of GPT-3.5

in identifying specific code smells (‘too many parameters’ and ‘too many nested blocks’), suggesting that further fine-tuning could enhance its stability.

Overall, even though GPT-3.5 demonstrates instability in code smell detection, our findings reveal that its precision and stability can be improved with further fine-tuning.

3.2 RQ2: Which code smells does GPT-3.5 effectively detect?

Based on Tables 3.1, 3.2, and 3.3, we observe that GPT-3.5 performs well in detecting the ‘too many parameters’ code smell. Our fine-tuned model exhibits high recall in identifying true positives. Although the model occasionally generates false positives, its F1-score of 0.77 indicates reliability in detecting ‘too many parameters’.

In the case of ‘unused variable’ detection, our fine-tuned model excels in true positives with a perfect recall of 1.00. However, its F1-score is only 0.34, suggesting unreliability in this detection task. For the ‘too many nested blocks’ smell, we find that GPT-3.5 still requires improvements in both precision and stability.

Observation 4: While GPT-3.5 effectively detects the ‘too many parameters’ code smell, it still needs to improve its reliability in identifying ‘too many nested blocks’ and ‘unused variables’.

3.3 RQ3: What impact does the size dataset have on the performance of GPT-3.5? in code smell detection?

To examine the influence of dataset size, we resize our dataset as shown in Figure 3.2. Since the smell ”too many nested blocks” is less common compared to the other two smells, we only reduce the dataset size for ‘too many nested blocks’ to 85% of its original size. We then adjust the sizes of the datasets for the other smells and non-smells to be similar.

We fine-tune a new GPT-3.5 model using the resized dataset. Tables 3.4, 3.5, and 3.6 present the performance comparison of models fine-tuned with different dataset sizes. Below is the summary of the change of GPT-3.5’s performance in resizing dataset.

- **Too many parameters:** The model increases to 0.96 in precision and decreases to 0.47 in recall, which means it obtains better capability in identifying non-smelly code but performs worse in detecting smelly code. Its F1 score reduces from 0.77 to 0.64, indicating that the model has less reliability.
- **Too many nested blocks:** The model exhibits a slight reduction in precision (0.54) but a notable increase in recall (0.98). This outcome indicates improved performance in code smell detection. The F1 score rises to 0.70, suggesting that the model is more reliable compared to the model with the full-size dataset.
- **Unused variable:** The model attains a precision of 0.5, a recall of 0.0, and an F1 score of 0.02, demonstrating its failure to detect ’unused variable’ smells.

Overall, our analysis reveals that dataset size significantly affects GPT-3.5’s overall performance. The model’s effectiveness declines in detecting ‘too many parameters’ and ‘unused

Table 3.4: Precision of the models with different dataset sizes

	Full-size dataset	Resized dataset	Change
Too Many Parameters	0.63	0.96	+0.33
Too Many Nested Blocks	0.59	0.54	-0.05
Unused Variable	0.20	0.50	+0.30

variable', while it shows notable improvement in identifying 'too many nested blocks'. Particularly, in detecting 'unused variable', the F1 score plummets from 0.34 to 0.02 with the resized dataset, suggesting it almost fails to identify any 'unused variable' issues.

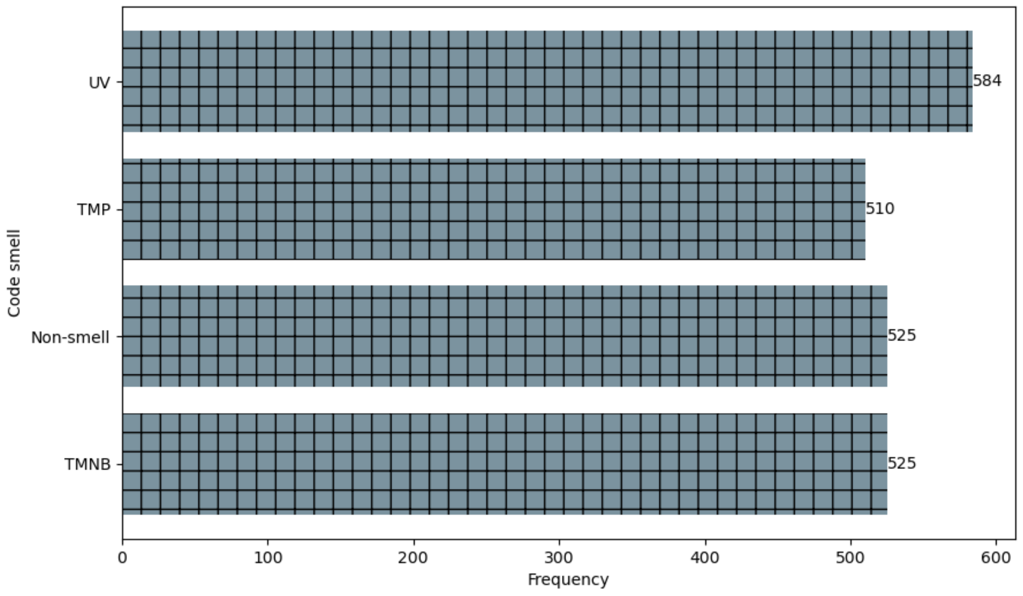


Figure 3.2: Data distribution after resizing dataset (**TMP**: too many parameters, **TMNB**: too many nested blocks, **UV**: unused variable)

Observation 5: The performance of GPT-3.5 is significantly influenced by the size of the dataset.

Table 3.5: Recall of the models with different dataset sizes

	Full-size dataset	Resized dataset	Change
Too Many Parameters	1.00	0.47	-0.53
Too Many Nested Blocks	0.39	0.98	+0.59
Unused Variable	1.00	0.00	-1.00

Table 3.6: F1 score of the models with different dataset sizes

	Full-size dataset	Resized dataset	Change
Too Many Parameters	0.77	0.64	-0.13
Too Many Nested Blocks	0.47	0.70	+0.23
Unused Variable	0.34	0.02	-0.32

3.4 Threats to Validity

The primary threat to the external validity of our evaluations is the dataset size. While we have created a comprehensive dataset from multiple projects and performed fine-tuning with varying dataset sizes to explore the impact, our dataset remains smaller compared to those commonly used for training large language models (LLMs). Therefore, our dataset needs further expansion.

The primary threat to the internal validity of our evaluations is the accuracy of Pylint. Pylint may produce false positives or false negatives, and we do not manually verify its results for the entire dataset due to our lack of expertise in software projects and the substantial effort required for manual checking. Additionally, we only assess three code smells. The evaluation would be more robust if we could extend it to include more code smells.

Chapter 4

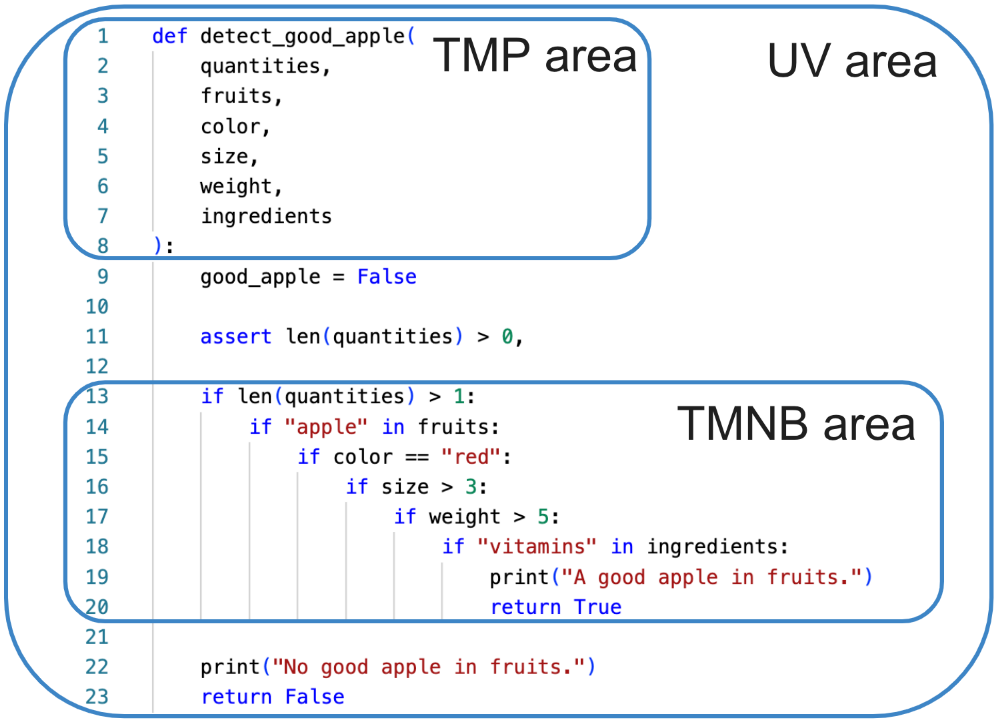
Discussion

In this section, we will discuss the conjectures from the results presented in the previous section.

4.1 Discussion 1: Is GPT-3.5's performance correlated to the context length of code smells?

Figure 4.1 presents an example function, "detect_good_apple", which exhibits all three identified code smells. This example illustrates the detection areas required for each type of code smell. The 'unused variable' smell demands the most extensive context, encompassing the entire function. The 'too many nested blocks' smell requires a more focused context, limited to the area containing the if statement. The 'too many parameters' smell needs the least context, only the line where the function is defined. Table 4.1 presents the context length ranking of the three code smells from high to low.

To explore the relation between context length and the complexity of code smells, we use



```python
def detect_good_apple(
    quantities,
    fruits,
    color,
    size,
    weight,
    ingredients
):
    good_apple = False

    assert len(quantities) > 0,

    if len(quantities) > 1:
        if "apple" in fruits:
            if color == "red":
                if size > 3:
                    if weight > 5:
                        if "vitamins" in ingredients:
                            print("A good apple in fruits.")
                            return True

    print("No good apple in fruits.")
    return False
```

Figure 4.1: Code smell context areas (**TMP**: too many parameters, **TMNB**: too many nested blocks, **UV**: unused variable)

Table 4.1: Rank of three code smells based on context length(from high to low)

Rank	Code Smell
1	Unused Variable
2	Too Many Nested Blocks
3	Too Many Parameters

Radon [18] to calculate three code metrics: cyclomatic complexity, maintainability index, halstead difficulty, and halstead effort. Below is a summary of each metric's function:

- **Cyclomatic Complexity:** Evaluates the number of linearly independent paths within a program's source code. This metric aids in assessing the complexity of a function or method, providing insights into its potential difficulty in testing and maintenance. Higher values suggest increased complexity and a greater likelihood of errors.

- **Maintainability Index:** This method combines multiple metrics, including cyclomatic complexity, halstead volume, and lines of code, to generate a unified score reflecting code maintainability. Higher scores signify greater maintainability.

- **Halstead Difficulty:** Assesses the complexity involved in writing or comprehending the program.
- **Halstead Effort:** Indicates the level of effort needed to develop or comprehend the code.

Table 4.2 presents the metrics for three code smells. The 'unused variable' consistently exhibits the highest values in cyclomatic complexity (8), halstead difficulty (1.5), and halstead effort (128.76). It also has the lowest maintainability index (61.08). These results indicate that 'unused variable' is the most complex code smell compared to the other two. Table 4.2 also shows that 'too many nested blocks' is the second most complex code smell, and 'too many parameters' is the least complex.

This pattern aligns with the code lengths observed in Figure 4.1. This correlation is evident because the larger the code context required by the code smell, the more context must be comprehended and taken into account.

Table 4.2: Complexity metrics of code smells (**CC:** cyclomatic complexity, **MI:** maintainability index, **HM (D):** halstead difficulty, **HM (E):** halstead effort)

	CC	MI	HM (D)	HM (E)
Unused Variable	**8**	**61.08**	**1.5**	**128.76**
Too Many Nested Blocks	7	64.31	1.5	105.49
Too Many Parameters	1	100	0	0

Conjecture 1: The difficulty of identifying code smells is associated to their context length.

According to the F1 score shown in Table 3.3 and the rank shown in Table 4.1, we find that the performance of GPT-3.5 is related to the complexity and context length of code smells. This relation is illustrated in Figures 4.2.

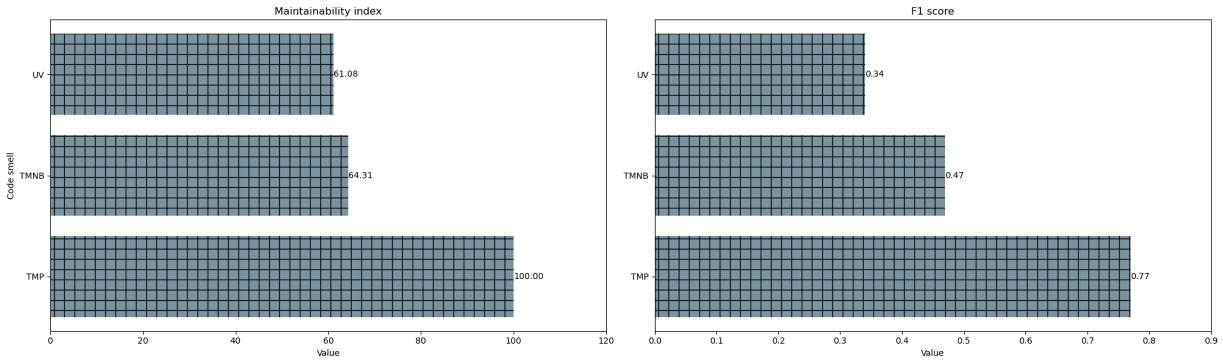


Figure 4.2: Relation between maintainability index and F1 score (**TMP**: too many parameters, **TMNB**: too many nested blocks, **UV**: unused variable)

Conjecture 2: GPT-3.5 is effective at identifying code smells in shorter contexts; however, its precision and stability diminish when dealing with longer contexts.

4.2 Discussion 2: Does increasing the amount of data for fine-tuning lead to more false positives in GPT-3.5?

Based on the results shown in 3, we find a relation between precision and recall. As illustrated in Figure 4.3, we analyze the precision and recall of the zero-shot model, the model fine-tuned with the resized dataset, and the model fine-tuned with the full-size dataset. We observe that the trend of precision and recall is asymmetrical. Subsequently, we calculate the Pearson correlation coefficient of precision and recall, as displayed in Table 4.3. The results indicate a negative correlation between precision and recall. This implies that when we fine-tune GPT-3.5 to detect code smells, as its performance in identifying smelly code improves, its performance in detecting non-smelly code diminishes.

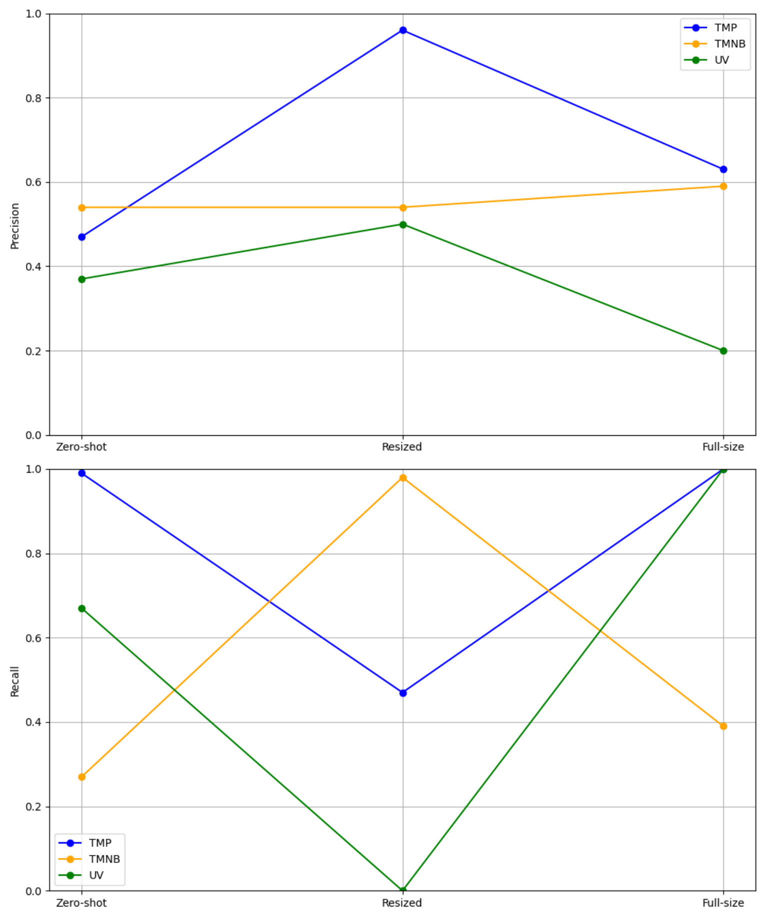


Figure 4.3: Precision (up) and recall (down) trend (**TMP:** too many parameters, **TMNB:** too many nested blocks, **UV:** unused variable)

Conjecture 3: Increasing the amount of data used to fine-tune GPT-3.5 leads to more false positives, despite its improved ability to identify smelly code.

4.3 Discussion 3: Will training sequence influence the performance of GPT-3.5?

As mentioned in 2, we fine-tune GPT-3.5 three times. During the first fine-tuning, we train it to detect Python code with 'too many parameters' and 'unused variable' smells. We find that GPT-3.5 can distinguishes between smelly and non-smelly code. However, after

Table 4.3: Pearson correlation coefficient of precision and recall

	Pearson correlation coefficient
Unused Variable	-0.94
Too Many Nested Blocks	-0.36
Too Many Parameters	-0.96

the second fine-tuning, which focuses on training it to detect non-smelly Python code, the results show all false positives, meaning it identifies all code as non-smelly. Finally, after the third fine-tuning, GPT-3.5 detects smelly code but fails to identify non-smelly code correctly.

Conjecture 4: Training sequence may effect GPT-3.5's performance.

Chapter 5

Conclusion and Future Work

In this study, we explore the efficacy of GPT-3.5 in detecting code smells, focusing on three smells in Python: too many parameters, too many nested blocks, and unused variable. Through comprehensive experiments involving 14 open-source software projects for fine-tuning and four additional projects for evaluation, our findings demonstrate that GPT-3.5 shows potential in code smell detection but still requires refinement and more data for fine-tuning.

The results indicate that GPT-3.5 is particularly effective in identifying the 'too many parameters' code smell, demonstrating reliable performance with a high F1 score of 0.77 post fine-tuning. However, its performance in detecting 'too many nested blocks' and 'unused variables' is less consistent, with F1 scores of 0.47 and 0.34, respectively, indicating a need for improvement in both precision and stability. Our research also reveals that dataset size significantly impacts GPT-3.5's performance in fine-tuning. Additionally, based on the results, we present several conjectures, including the relation between GPT-3.5's performance and the context length of code smells, the relation between dataset size and false positives, and the influence of training sequences.

Our future work involves incorporating a broader range of open-source projects, including additional code smells and projects written in various programming languages. This expansion helps to generalize the model's applicability and performance across diverse codebases and environments. Another direction involves exploring and refining additional factors that may influence GPT-3.5's performance during fine-tuning, such as temperature and steps.

Bibliography

[1] X. Amatriain. Prompt design and engineering: Introduction and advanced methods. *arXiv preprint arXiv:2401.14423*, 2024.

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[3] Checkstyle. Checkstyle. `https://checkstyle.sourceforge.io/`, 2015.

[4] A. K. Das, S. Yadav, and S. Dhal. Detecting code smells using deep learning. In *TENCON 2019-2019 IEEE Region 10 Conference (TENCON)*, pages 2081–2086. IEEE, 2019.

[5] F. A. Fontana, P. Braione, and M. Zanoni. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.*, 11(2):5–1, 2012.

[6] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[7] L. Giray. Prompt engineering with chatgpt: a guide for academic writers. *Annals of biomedical engineering*, 51(12):2629–2633, 2023.

[8] A. Gupta, R. Gandhi, N. Jatana, D. Jatain, S. K. Panda, and J. V. N. Ramesh. A severity assessment of python code smells. *IEEE Access*, 2023.

[9] A. Gupta, B. Suri, and S. Misra. A systematic literature review: code bad smells in java source code. In *Computational Science and Its Applications–ICCSA 2017: 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part V 17*, pages 665–682. Springer, 2017.

[10] R. Gupta and S. K. Singh. A novel transfer learning method for code smell detection on heterogeneous data: A feasibility study. *SN Computer Science*, 4(6):749, 2023.

[11] S. Kublik and S. Saboo. *GPT-3*. O'Reilly Media, Incorporated, 2022.

[12] Logilab. Pylint. `https://pylint.org/`, 2001.

[13] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 350–359. IEEE, 2004.

[14] OpenAI. Models. `https://platform.openai.com/docs/models`, 2024.

[15] OpenAI. Preparing your dataset. `https://platform.openai.com/docs/guides/fine-tuning/preparing-your-dataset`, 2024.

[16] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278. IEEE, 2013.

[17] PMD. Pmd. `https://pmd.github.io/`, 2015.

[18] F. Rubik. Radon. `https://github.com/rubik/radon`, 2023.

[19] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018.

[20] W. C. Wake. *Refactoring workbook*. Addison-Wesley Professional, 2004.

[21] M. Wiltzer. Refactoring the nested conditionals code smell. `https://makolyte.com/refactoring-the-nested-conditionals-code-smell/`, 2024.