Leveraging Prompt-based Large Language Models for Code Smell Detection: A Comparative Study on the MLCQ Dataset

Djamel Mesbah, Nour El Madhoun, Khaldoun Al Agha, Hani Chalouati

▶ **To cite this version:**

Djamel Mesbah, Nour El Madhoun, Khaldoun Al Agha, Hani Chalouati. Leveraging Prompt-based Large Language Models for Code Smell Detection: A Comparative Study on the MLCQ Dataset. The 13-th International Conference on Emerging Internet, Data & Web Technologies (EIDWT-2025), Feb 2025, Matsue, Japan. hal-04881949

HAL Id: hal-04881949

https://hal.science/hal-04881949v1

Submitted on 13 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Leveraging Prompt-based Large Language Models for Code Smell Detection: A Comparative Study on the MLCQ Dataset

Djamel Mesbah[123], Nour El Madhoun[34], Khaldoun Al Agha[2], and Hani Chalouati[1]

[1] Adservio Group, T. Franklin, 100 101 Terr. Boieldieu Ét. 9, 92800, Puteaux, France
{djamel.mesbah,hani.chalouati}@adservio.fr
[2] Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, 91190, Gif-sur-Yvette, France
djamel.mesbah@universite-paris-saclay.fr, alagha@lisn.fr
[3] LISITE Laboratory, Isep, 10 Rue de Vanves, 92130, Issy-les-Moulineaux, France
{djamel.mesbah, nour.el-madhoun}@isep.fr
[4] Sorbonne Université, CNRS, LIP6, 4 place Jussieu, 75005, Paris, France
nour.el_madhoun@sorbonne-universite.fr

Abstract. Code smells are indicators of potential issues in software code that can make maintenance more challenging. Traditional approaches to detecting code smells have primarily relied on handcrafted rules and heuristics, while recent advances have explored Machine Learning (ML) and Deep Learning (DL) techniques. In this paper, we investigate the application of prompt-based Large Language Models (LLMs) for code smell detection, utilizing state-of-the-art models, namely Generative Pre-trained Transformer-4 (GPT-4) and Large Language Model Meta AI (LLaMA). We conduct an extensive analysis of the Machine Learning Code Quality (MLCQ) dataset, focusing on how these LLMs perform when prompted to identify and classify code smells. By systematically evaluating each model's performance, we provide insights into their precision, recall and ability to generalize across different types of code smells. Our results aim to demonstrate the potential of LLMs as a promising tool for automating certain types of code smells while underperforming for others.

Keywords: Code Smells · GPT-4 · Large Language Models · LLaMA · LLMs · Machine Learning

1 Introduction

Code smells are indicators of potential issues in the codebase that suggest a need for refactoring. Their presence affects the readability, maintainability and extensibility of the source code. Early detection of code smells is therefore crucial for the software evolution process, helping to reduce the long-term costs associated with refactoring. The detection of code smells has been an area of

extensive study. Traditional approaches primarily rely on heuristics and manually crafted metrics to classify code elements as "smelly" or "non-smelly" [1]. However, these metrics often lack a consensus among developers [2, 3], leading to inconsistent detection outcomes. To address this, recent research has shifted toward ML methods, particularly supervised learning, for automated smell detection.

In the ML-based approach, models are trained on labeled data to classify code segments such as methods or classes—based on their likelihood of containing code smells. This method can significantly reduce manual effort by leveraging large datasets to train classifiers capable of automatically identifying relevant metrics and establishing thresholds for smell detection. The features provided to these classifiers can include standard code metrics (for example: Lines Of Code (LOC) and Number Of Attributes (NOA)) [4], token sequences extracted from the code [5], or graph-based representations capturing the semantic structure of the code such as Abstract Syntax Trees (ASTs) [6, 7].

In this paper, we investigate the use of prompting techniques with LLMs specifically GPT and LLaMA, to detect code smells in the Machine Learning Code Quality (MLCQ) dataset. We evaluate their ability to identify smells and their severity. The code is available at [1]

The paper is organized as follows. Section 2 provides an overview of related work, while Section 3 presents a background on code smells and ML in the software landscape. Section 4 describes the dataset and the process followed. Section 5 discusses the results. Finally, Section 6 concludes the study.

2 Related Work

Code smell detection has traditionally been approached through heuristic methods, where specific characteristics or patterns in code, defined by expert-crafted rules, indicate the presence of a smell. For instance, approaches such as DECOR [1] relied on manually defined thresholds for code metrics like complexity or coupling to classify code as "smelly" or "non-smelly." However, the need for external intervention to define these thresholds and rules presents limitations, as code smell detection is inherently subjective and context-dependent.

With the rise of ML, there was a shift towards using statistical models trained on code metrics. Techniques like decision trees and Random Forests classified code snippets based on features such as LOC, cyclomatic complexity and coupling measures [4]. Although these methods provided more flexibility than purely heuristic approaches, they still relied on engineered features, which might not fully capture the semantic richness of source code.

The adoption of Deep Learning (DL) represented a more significant shift, moving from feature-based classification to token-level processing. Models, such as Convolutional Neural Networks (CNNs) [8], Long Short-Term Memory networks (LSTMs) [5] and fully connected networks, were used to process sequences

[1] `https://github.com/Kheims/llm_mlcq.git`

of tokens from code snippets, thereby learning complex patterns directly from the source code. This approach provided more accurate representations of the code's structure and semantics, exceeding the capabilities of traditional ML.

Recently, the emergence of LLMs, such as Bidirectional Encoder Representations from Transformers (BERT), GPT-3, GPT-4 and Claude, pretrained on massive datasets including both text and code, has opened new avenues for code-related tasks. These models excel in understanding and generating code [9–11], suggesting that they could potentially be adapted for detecting code smells as well.

The adoption of LLMs in code-related tasks has opened new possibilities for code smell detection. Building on the capabilities of models like BERT and GPT-4, recent approaches explore ways to leverage LLMs' semantic understanding to identify subtle design issues in code. For instance, [12] introduces PromptSmell, a method that utilizes prompt-based learning combined with Abstract Syntax Tree (AST) traversal to detect code smells in Java. By framing code snippets as natural language prompts and translating model outputs to predefined smell categories through a verbalizer, PromptSmell effectively addresses multi-label classification challenges in code quality assessment. Their results demonstrate that this prompt-based approach outperforms traditional and fine-tuned models in both accuracy and F1 scores.

Similarly, [13] assesses LLMs like GPT-4, Gemini Advanced and Mistral Large in identifying test smells across multiple programming languages. This study underscores the efficiency of LLMs in capturing a wide range of test smells, with GPT-4 identifying up to 70% of smell types. These implementations show that by directly leveraging LLMs' rich contextual understanding, it is possible to streamline the detection of both code and test smells, minimizing the reliance on handcrafted rules and manual intervention.

3 Background

Code smells, originally introduced by Kent Beck [2] within the context of refactoring, are indicators in the code that signal potential issues, though they do not necessarily imply an immediate malfunction. Unlike bugs, code smells don't prevent the software from working, but they often point to suboptimal design choices or poor coding practices. These smells can degrade code readability, introduce unnecessary complexity and make the code more difficult to maintain. Detecting code smells is critical because they contribute to making the code harder to comprehend and modify, thereby increasing the risk of errors and future bugs. These smells can be of three types [14] : implementation, design and architectural smells all signaling problems at different levels of the code base.

Traditionally, detecting code smells relied on heuristics and manually defined rules derived from expert knowledge [1], which made these approaches dependent on human intervention and prone to subjectivity. With the rise of ML and DL,

[2] `https://martinfowler.com/bliki/CodeSmell.htmlr`

there has been a shift towards automating code smell detection by leveraging statistical methods and language models.

LLMs have recently emerged as powerful tools for a variety of natural language processing tasks, including code-related tasks like code generation and analysis. LLMs are a class of DL models that rely on self-attention mechanisms to capture dependencies between tokens in sequences of text, including code [15]. Their ability to process vast amounts of data has made them the leading approach for language tasks. LLMs differ from traditional language models in terms of their size and the amount of data they are trained on, allowing them to capture more intricate relationships in text.

We note three types of LLMs based on their architecture :

- Encoder-only LLMs (for example: BERT [16]): these models use only the encoder module to transform input text into hidden representations. Encoder-only models capture word relationships and contextual information using bidirectional attention, allowing the model to consider both the preceding and following context of each word. These models excel in tasks like text classification and code comprehension.
- Encoder-decoder LLMs [17]: these models consist of an encoder that processes the input and a decoder that generates output. The encoder converts input text into a hidden representation, which serves as a bridge between input and output formats. The decoder uses this hidden space to generate coherent and contextually relevant output text, making these models suitable for tasks like translation or summarization.
- Decoder-only LLMs (for example: the GPT series [18]): these models focus on generating output text in an autoregressive manner, predicting each token one at a time based on the preceding tokens. Decoder-only models are particularly powerful for text and code generation tasks due to their ability to handle long sequences and generate coherent output.

Given the rise of LLMs and their emergent behavior at excelling in working on code-related tasks such as code generation [10], code summarization [19] and bug localization [20], we hypothesize that their application to code smell detection could provide significant advantages. Unlike traditional methods, LLMs can learn the complex patterns and semantics of code directly from data without relying on manually designed rules, potentially making them more effective at detecting subtle smells and predicting their severity. For this intent, we aim to explore the effectiveness of using GPT-4 as well as LLaMA 3.

GPT-4 is a large-scale AI-based text generator built upon the InstructGPT architecture [21]. InstructGPT introduced Reinforcement Learning with Human Feedback (RLHF), enhancing the model's ability to understand human intent beyond conventional next-token prediction. Through this combination of instruction tuning and conversational format conversion, GPT-4 has demonstrated significant improvements in various natural language processing tasks. It produces text outputs by leveraging a larger scale of compute, following power laws to optimize its training on vast datasets. This makes GPT-4 an advanced model

not only for text generation but also for tasks requiring nuanced reasoning and contextual understanding [22].

LLaMA 3 [23], developed by Meta, is an open-source LLM that natively supports multilingual tasks, coding, reasoning and tool usage. The largest LLaMA model features 405 billion parameters and a context window extending up to 128K tokens, with significant improvements in inference scalability through the Grouped Query Attention (GQA) mechanism [24]. LLaMA comes in various sizes and for this study, we focus on the 8B Instruct version—utilized in both 16-bit and 8-bit precision modes.

Meta's hardware recommendations for running the LLaMA 3.1 8B model include a modern processor with at least 8 cores and a minimum of 16 GB of RAM. For GPU requirements, Meta suggests the NVIDIA RTX 3090 or RTX 4090 (24 GB VRAM) for models in 16-bit mode. The estimated disk space required is approximately 20-30 GB for the model and associated data, with GPU memory usage varying by precision, specifically for the 32-bit mode: 38.4 GB of memory is needed, whereas the 16-bit mode halves the memory requirement to 19.2 GB. Due to hardware constraints, we conducted our experiments on the 8B Instruct LLaMA model for both the 16 bit mode and the 8 bit mode, running the inference on a system with 2 x Intel Xeon Gold 6148 (20 cores / 40 threads @ 2.4 GHz Skylake), 384 GiB of RAM and a NVIDIA Tesla V100 GPU with 32 GiB of RAM (NVLink).

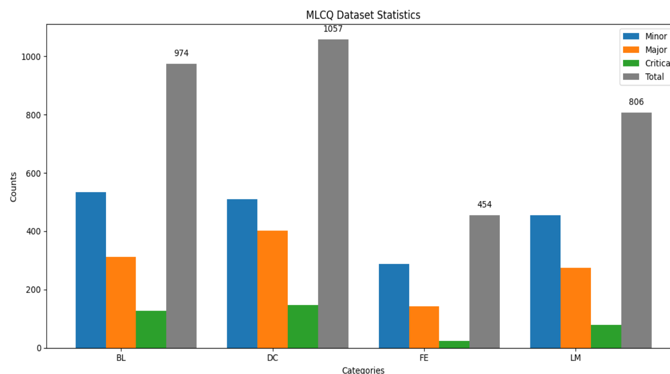

Fig. 1: MLCQ statistics

4 Dataset and Process

4.1 Dataset

Our work studies the applicability of smell detection through LLMs for the MLCQ dataset [25], an industry dataset comprising nearly 15000 samples created by expert software developers who reviewed industry-relevant Java open source projects. The dataset contains for each sample the smell found between 4 implementation smells: Blob Class, Data Class, Feature Envy and Long Method

and severity for each smell ranging from none, minor, major and critical (see Fig. 1). In addition the developers went through an extensive survey to fill in multiple pieces of information on their professional experience making it a reliable dataset to study in comparison to datasets solely made through the use of heuristics tools that may introduce inherent bias.

Prompt Structure: Vanilla Prompt for Code Smell Detection

```
You are a code analysis assistant. Please analyze the
following code snippet and identify any code smell between:
"feature_envy", "long_method", "blob", "data_class".

Additionally, rate the severity of the code smell as:
"none", "minor," "moderate," or "severe."

Code snippet: {code_snippet}

Provide your response in the exact format:
"Smell: <name>, Severity: <severity>"

Do not add any other information to the response.
```

(a) Vanilla Prompt Structure

Few Shot Structure: Vanilla Prompt for Code Smell Detection

```
prompt = "You are a code analysis assistant. Below are examples of code snippets
with identified code smells and severity.
Use this information to analyze the next
code snippet and identify any
code smell between:\n"
prompt += ’"feature_envy", "long_method", "blob", "data_class".
Additionally, rate the severity of the code smell as: "none", "minor", "moderate",
or "severe."\n\n’

prompt += "Examples:\n"

for i, example in enumerate(examples, start=1):
    prompt += f"{i}. Code snippet:\n‘‘‘\n{example[’code_snippet’]}\n‘‘‘\n"
    prompt += f"Smell: {example[’smell’]}, Severity: {example[’severity’]}\n\n"

prompt += "Now, analyze the following code snippet:\n\n"
prompt += "Code snippet:\n‘‘‘\n" + code_snippet_to_analyze + "\n‘‘‘\n"
prompt += ’Provide your response in the exact format: "Smell: <name>, Severity: <severity>"’.
```

(b) Few Shot Prompt Structure

Fig. 2: Comparison of Prompt Structures for Code Smell Detection

4.2 Prompts

Our process to define the prompts follows an iterative process where we first provide the models with a vanilla prompt, specifying the range of smells that can be found as well as their range of severity and enforcing the result schema. The summary of the prompt can be seen in Fig. 2a. Then, we incorporate examples into the prompts for few-shot learning in order to assess the model's ability to specialize in a specific task and whether it is indeed effective. The prompt can be seen in Fig. 2b.

4.3 Evaluation Metrics

To assess the LLMs' performance in detecting smells we apply the widely used metrics precision, recall and F1 measure.

- **Precision** (P) is the proportion of predicted positive cases that are correctly identified:

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

- **Recall** (R), also known as sensitivity or true positive rate, is the proportion of actual positive cases that are correctly identified:

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

- **F1 Score** (F1) is the harmonic mean of precision and recall, which provides a balance between the two:

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

In this context, precision measures the proportion of correctly identified smelly snippets among all snippets classified as smelly, indicating the model's ability to avoid false positives. Recall, on the other hand, evaluates the proportion of correctly identified smelly snippets out of all actual smelly snippets, reflecting the model's ability to minimize false negatives.

5 Results

To better understand the contribution of different components to the performance of our models, we conducted a study focusing on the following aspects:

- Prompting Style (Vanilla vs. Few-shot): We assessed the impact of including few-shot learning examples in the prompt. Table 1 shows the relative improvement in precision, recall and F1 score across all models when transitioning from vanilla to few-shot prompting.

– Model Quantization (16-bit vs 8-bit): We analyzed the trade-off between computational efficiency and performance for LLaMA models by comparing the quantized 16-bit version with the quantized 8-bit version. Tables 1 and 2 show that the loss in performance is minimal while quantization significantly reduces memory usage and speeds up inference.

The overall performance of the models, as indicated by the precision, recall and F1 scores, suggests that code smell detection using LLMs is still a challenging task. The models underperform relative to what would be expected for practical code analysis, with most metrics falling below the level needed for reliable detection. GPT-4, as the best performer, demonstrates its ability to closely follow the prompt and produce responses that are well-aligned with the expected format. This is especially true in terms of its adherence to the schema for smells and severity. GPT-4 achieves higher precision and F1 scores than LLaMA, showing its general capability for instruction-following and task-specific adaptation. However, the performance still remains low in absolute terms, suggesting the complexity of the task and the inherent difficulty in recognizing and classifying code smells correctly.

In contrast, LLaMA presents a unique challenge. When using the vanilla prompt, it often provides extensive, detailed analyses of the code, focusing on logical and semantic interpretations rather than adhering strictly to the task of identifying smells and severity. This behavior points to the model's tendency to over-explain and generate exhaustive outputs, which, while informative, do not align with the required concise answers. This necessitates substantial post-processing to extract relevant information and align it to the desired structure.

The few-shot prompting approach significantly enhances the model's performance, particularly for GPT-4, where a marked improvement in detecting smells such as long method and data class is observed. Few-shot examples seem to enable GPT-4 to generalize better from limited supervision, improving its ability to handle more complex smells. The F1 scores increase substantially, indicating that GPT-4 can learn from examples and apply this learning to new instances.

For LLaMA, few-shot prompting proves to be a game-changer. The model transitions from providing verbose, loosely aligned answers to generating more concise, schema-conforming outputs although the results are still afar from GPT-4 providing reasonable results only for the data class smell. LLaMA's quantized 8-bit version, despite offering some computational benefits, exhibits a slight drop in performance compared to its 16-bit counterpart. This trade-off is expected in exchange for faster inference times and reduced memory requirements.

Table 1: Comparison of Precision, Recall and F1-Measure with Vanilla Prompt

Smell	GPT-4			LLaMA (Quantized 16-bit)			LLaMA (Quantized 8-bit)		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
Feature Envy	0.143	0.02	0.035	0.09	0.62	0.157	0.081	0.558	0.141
Long Method	0.44	0.78	0.562	0.20	0.20	0.20	0.18	0.18	0.18
Blob	0.25	0.006	0.012	0.06	0.001	0.002	0.054	0.001	0.002

Table 2: Comparison of Precision, Recall and F1-Measure with Few-Shot Prompting

Smell	GPT-4			LLaMA (Quantized 16-bit)			LLaMA (Quantized 8-bit)		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
Feature Envy	0.172	0.03	0.051	0.11	0.68	0.19	0.095	0.62	0.165
Long Method	0.62	0.85	0.717	0.24	0.24	0.24	0.20	0.20	0.20
Blob	0.30	0.008	0.015	0.072	0.002	0.003	0.065	0.0011	0.002
Data Class	0.765	0.78	0.772	0.70	0.09	0.16	0.648	0.075	0.134

6 Conclusion

In this study, we explored the application of LLMs, specifically GPT-4 and LLaMA, for the task of code smell detection, using the MLCQ dataset. Our results indicate that while LLMs have shown promise in various code-related tasks, their performance on code smell detection remains limited, often underperforming when compared to the expectations set by traditional ML models. GPT-4, particularly with the use of few-shot prompting, outperformed other models, demonstrating its capacity to closely follow task-specific instructions and provide more reliable outputs. However, its overall performance still suggests room for improvement in terms of precision and generalization.

LLaMA, particularly in its vanilla form, exhibited a tendency to generate verbose, logically detailed responses that did not always align with the specific task requirements. However, when augmented with few-shot prompting, LLaMA demonstrated a significant improvement in adhering to the task schema and providing more accurate classifications. The quantized 8-bit version of LLaMA showed a slight degradation in performance, while still maintaining the advantage of reduced memory usage and faster inference.

Despite these advancements, several challenges remain, particularly in addressing class imbalance, improving the detection of nuanced code smells and enhancing severity classification. The models struggled with feature envy and blob detection in particular, where F1 scores remained low even with prompting techniques. This highlights the complexity of detecting certain smells and suggests that further fine-tuning and model-specific adaptations are required for effective detection in real-world scenarios.

References

1. Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2009.
2. Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. Antipattern and code smell false positives: Preliminary conceptualization and classification. *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), IEEE*, 1:609–613, 2016.
3. Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.*, 11(2):5–1, 2012.
4. Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. Comparing heuristic and machine learning approaches for metric-based code smell detection. in 2019 ieee/acm 27th international conference on program comprehension (icpc). *IEEE, 93ś104*, 2019.
5. Anh Ho, Anh MT Bui, Phuong T Nguyen, and Amleto Di Salle. Fusion of deep convolutional and lstm recurrent neural networks for automated detection of code smells. *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 229–234, 2023.
6. Yichen Li and Xiaofang Zhang. Multi-label code smell detection with hybrid model based on deep learning. *SEKE*, pages 42–47, 2022.
7. Weiwei Xu and Xiaofang Zhang. Multi-granularity code smell detection using deep learning method based on abstract syntax tree. *SEKE*, pages 503–509, 2021.
8. Tao Lin, Xue Fu, Fu Chen, and Luqun Li. A novel approach for code smells detection based on deep leaning. *Applied Cryptography in Computer and Communications: First EAI International Conference, AC3 2021, Virtual Event, May 15-16, 2021, Proceedings 1, Springer*, pages 171–174, 2021.
9. Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
10. Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.
11. Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology, ACM New York, NY*, 2023.
12. Haiyang Liu, Yang Zhang, Vidya Saikrishna, Quanquan Tian, and Kun Zheng. Prompt learning for multi-label code smell detection: A promising approach. *arXiv preprint arXiv:2402.10398*, 2024.
13. Keila Lucas, Rohit Gheyi, Elvys Soares, Márcio Ribeiro, and Ivan Machado. Evaluating large language models in detecting test smells. *arXiv preprint arXiv:2407.19261*, 2024.
14. Djamel Mesbah, Nour El Madhoun, and Khaldoun Al Agha. Beyond the code: Unraveling the applicability of graph neural. *Advances in Network-Based Information Systems: The 27th International Conference on Network-Based Information Systems (NBiS-2024), Springer Nature*, page 148.

15. A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.
16. Mikhail V Koroteev. Bert: a review of applications in natural language processing and understanding. *arXiv preprint arXiv:2103.11943*, 2021.
17. Zihao Fu, Wai Lam, Qian Yu, Anthony Man-Cho So, Shengding Hu, Zhiyuan Liu, and Nigel Collier. Decoder-only or encoder-decoder? interpreting language model as a regularized encoder-decoder. *arXiv preprint arXiv:2304.04052*, 2023.
18. Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.
19. Rajarshi Haldar and Julia Hockenmaier. Analyzing the performance of large language models on code summarization. *arXiv preprint arXiv:2404.08018*, 2024.
20. Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. A deep dive into large language models for automated bug localization and repair. *Proceedings of the ACM on Software Engineering, ACM New York, NY, USA*, 1(FSE):1471–1493, 2024.
21. Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.
22. Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
23. Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
24. Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.
25. Lech Madeyski and Tomasz Lewowski. Mlcq: Industry-relevant code smell data set. *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, pages 342–347, 2020.