Java Code Optimization using LLMs

Nishtha Aggarwal (0028916709)
Purdue University
West Lafayette, Indiana, USA
naggarwal@purdue.edu

Jiangqiong Liu (0033762621)
Purdue University
West Lafayette, Indiana, USA
liu3328@purdue.edu

Mayesha Monjur (0035773359)
Purdue University
West Lafayette, Indiana, USA
monjur@purdue.edu

ABSTRACT

In the contemporary landscape of software development, optimizing code for efficiency and performance is a paramount challenge that developers face. We introduce an innovative approach to code optimization by leveraging the capabilities of Large Language Models (LLMs), with a specific focus on Java and GraphCodeBERT [2].

The significance of this endeavor lies in its potential to revolutionize coding platforms such as LeetCode and HackerRank, by providing developers and aspiring coders with advanced problem-solving strategies and optimized coding solutions. Our strategy involves a prioritized exploration of three avenues: primary focus on pretraining GraphCodeBERT with a tailored task for optimization, an adaptive strategy for structured code output via GraphCodeBERT through iterative optimization refinement, and, as a final resort, the integration of few-shot learning and GPT models for optimization with minimal input.

This project not only aims to enhance the coding efficiency and performance for Java developers but also sets the stage for a new standard in automated code optimization. Through this initiative, we envision a future where AI-driven tools significantly elevate coding practices, leading to more streamlined, effective, and innovative software development processes.

CCS CONCEPTS

• **Software and its engineering → Software organization and properties**; • **Software functional properties → Software performance**; • **Computing Methodologies → Natural Language Processing**.

KEYWORDS

GraphCodeBERT, Few-Shot Learning, GPT, Restructuring and Optimization, Java

ACM Reference Format:
Nishtha Aggarwal (0028916709), Jiangqiong Liu (0033762621), and Mayesha Monjur (0035773359). 2018. Java Code Optimization using LLMs. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

In the rapidly advancing field of software development, the quest for efficiency and optimization is unending. The advent of Large Language Models (LLMs) has ushered in a revolutionary era, presenting novel avenues for automating and refining coding practices. Our research project is strategically positioned to capitalize on these advancements, with a singular focus on leveraging the potential of LLMs to transform code optimization for Java. This initiative is designed to empower our AI tool to pioneer innovative optimization strategies and methodologies, potentially revolutionizing platforms such as LeetCode and HackerRank.

The significance of this project extends far beyond simple code enhancement. It seeks to empower developers and aspiring coders by equipping them with advanced problem-solving strategies and optimized coding solutions, facilitating not only the interview process but also contributing to the overarching aim of elevating coding efficiency and performance. Our commitment is to utilize LLMs, with a specific emphasis on Java code implementations, heralding a new era in automated code refinement.

Our strategic plan is structured around a prioritized exploration of three cutting-edge avenues, meticulously organized based on their projected impact and feasibility:

(1) **Focused Pretraining with GraphCodeBERT:** Our primary strategy involves pretraining a model, specifically GraphCodeBERT, accompanied by a tailored pretraining task designed to ensure the output code is not merely functional but optimized. GraphCodeBERT is our chosen model due to its potential in code understanding and restructuring and the fact that it is pretrained on Java; however, our search for excellence is ongoing, and we are open to transitioning to a more effective model should one be discovered during our research journey.
(2) **Adaptive Structured Code Output as a Secondary Strategy:** Should our initial approach encounter obstacles or fail to achieve the desired outcomes, our secondary plan involves deploying GraphCodeBERT for the generation of restructured Java code. This process is uniquely iterative, involving the model re-evaluating and restructuring the code until it either meets a predefined performance improvement threshold or reaches a maximum number of iterations. This ensures a continuous refinement loop, providing multiple opportunities for the code to reach the desired level of optimization.
(3) **Exploratory Few-Shot Learning and GPT Integration as a Final Resort:** In scenarios where both the primary focus and the adaptive secondary strategy fall short of expectations, our last resort will be to delve into the integration of few-shot learning techniques and GPT models. This approach aims to refine the model's ability to output optimized Java code with minimal input, capitalizing on the adaptive

learning prowess of GPT models for effective application of optimization strategies after exposure to a limited number of examples.

This comprehensive and adaptable methodology underscores our dedication to pushing the boundaries of code optimization using LLMs, specifically for Java. By outlining clear priorities and establishing a systematic approach to overcoming potential challenges, we aim not only to enhance our tool's capability to optimize code but also to set a new standard in coding efficiency that could significantly benefit the broader software development community.

2 RELATED WORK

In recent years, there has been a surge of interest in leveraging machine learning techniques, particularly large language models (LLMs), for optimization tasks. Several noteworthy papers have contributed to this field and explored various aspects of this domain.

The Supersonic model [3] automates optimization for C/C++ programs by targeting minor source code modifications for optimization and retaining significant similarity with the original program. The sequence-to-sequence model learns the relationship between input programs and their optimized versions, and it features an original diff-based output representation that is similar to a software patch. The 7-billion-parameter transformer model [1] is designed to take unoptimized assembly code as input and produce a list of compiler options that would best optimize the program for reduced code size. It demonstrates the effectiveness of leveraging LLMs for code optimization tasks and inspire the research community to push beyond LLMs for simple max-likelihood code generation and into performance-aware code optimization.

Besides specific architectures and implementations, frameworks for applying Large Language Models to optimization tasks are studied. The Optimization by PROmpting approach [5] generates solutions for optimization problem in natural language by instructing the LLM to iteratively generate new solutions based on the problem description and the previously found solutions. This approach is effective for mathematical optimization problems like the Traveling Salesman Problem and prompt optimization for finding the prompt that maximizes task accuracy. Also, the framework [4] proposes a range of adaptation strategies for code optimization, including various prompting techniques such as retrieval-based few-shot prompting and chain-of-thought, as well as finetuning approaches like performance-conditioned generation and synthetic data augmentation based on self-play. The performance-conditioned generation technique is very effective in improving program performance and increasing the fraction of optimized programs. The combination of these techniques yields significant improvements.

These papers collectively contribute to advancing the field of performance optimization by harnessing the power of large language models. They offer valuable insights, methodologies, and frameworks for automating code optimization tasks, paving the way for more efficient and scalable software development practices.

None of the papers referenced have delved into code optimization through pretraining tasks. We believe that leveraging pretraining tasks specifically tailored for code optimization holds promise for yielding superior results compared to alternative techniques. Additionally, there exists no code optimization solutions using Large Language Models (LLMs) tailored specifically for the language Java, which motivated us to work with Java given its prevalence and importance in various industrial applications. Our project pioneers Java code optimization by leveraging GraphCodeBERT, uniquely pretraining it with Java-specific tasks to refine code effectively. This method stands out for its potential to understand and optimize Java code beyond the capabilities of current models. Alternatively, we might work with an adaptive strategy for structured code output, incorporating a novel continuous refinement loop for iterative performance improvements. Facing further potential challenges, we're prepared to explore few-shot learning and GPT integration, enhancing our model's adaptability. This comprehensive approach, emphasizing Java-specific optimization, sets a new precedent in the application of LLMs for code optimization, promising enhanced efficiency and setting a benchmark for future research.

3 METHODOLOGY AND EVALUATION PLAN

To effectively evaluate the impact of each proposed approach on code optimization, we will implement a comprehensive evaluation plan. This plan aims to quantify performance improvements (PI) in terms of running time or memory usage between the original and the optimized Java programs. The PI is defined as PI = $\frac{\text{old}}{\text{new}}$, where 'old' and 'new' refer to the running time or memory usage of the original and optimized programs, respectively. The average PI across a designated test set will be reported for each approach.

3.1 Computation Resources

Our initial strategy for model training is to utilize Google Colab Pro. We anticipate that the computational capabilities provided by this platform will be adequate for our model's requirements. However, should we encounter limitations in training our model effectively on Google Colab Pro, we acknowledge the potential need for additional computational resources. In such an event, we plan to explore advanced computing infrastructures, specifically computers equipped with Graphics Processing Units (GPUs), to ensure the efficient and successful training of our model.

3.2 Pretraining GraphCodeBERT with a Tailored Task for Optimization

Objective: Assess the pretrained GraphCodeBERT model's effectiveness in optimizing Java code.
Methodology: Compile and execute a test set of Java programs to establish baseline performance metrics. Optimize the programs using the pretrained GraphCodeBERT model and evaluate the optimized versions.
Evaluation Metrics: Calculate the PI for each program and report the average PI across all programs. A higher average PI indicates greater optimization effectiveness.

3.3 Adaptive Strategy for Structured Code Output via GraphCodeBERT

Objective: Determine the performance improvement through iterative optimization refinement using GraphCodeBERT.

Methodology: Iteratively optimize each Java program in the test set, using each iteration's output as the next input, until a performance improvement threshold is met or a maximum number of iterations is reached.
Evaluation Metrics: Analyze the PI progression across iterations for each program and average the final PI values across the test set.

3.4 Integration of Few-Shot Learning and GPT Models for Optimization

Objective: Evaluate the optimization capacity of few-shot learning and GPT models on Java code with minimal input.
Methodology: Optimize a selection of Java programs using few-shot learning with GPT models, then execute the original and optimized versions to measure performance differences.
Evaluation Metrics: Compute the PI for each optimized program and average these values across the test set to assess the overall effectiveness.

In addition to quantitative metrics, qualitative aspects such as code readability, maintainability, and complexity post-optimization will also be considered in the evaluation. This robust plan ensures a thorough assessment of each strategy's potential to advance Java code optimization.

4 DATASET COLLECTION

Our objective in gathering a dataset is to compile a meticulously curated collection of Java code pairs, encompassing both unoptimized and optimized versions, for each coding question sourced from platforms like LeetCode and HackerRank to assist us with the pre-training task of Approach 1 in our methodology.

The creation of this dataset stands as a principal contribution of our paper, underscoring several significant challenges. Foremost among these challenges is the scarcity of publicly available datasets tailored precisely for this purpose. While there are datasets for code completion or code summarization tasks, there are no publicly available datasets containing pairs of unoptimized and optimized Java code snippets. Collecting a large number of Java code solutions for LLM training is a time-consuming and labor-intensive process. It requires manually reviewing and selecting appropriate code samples that cover a diverse range of programming challenges and optimization scenarios. Besides, pairing each unoptimized code snippet with its corresponding optimized version requires meticulous attention to detail. Ensuring that both versions maintain the same functionality while demonstrating the difference in optimization levels is crucial for the effectiveness of the dataset.

We have tried scraping such problems along with their solutions from two popular coding platforms - LeetCode and HackerRank. Our dataset would include around 3000-4000 problems and their solutions. The findings are following:

4.1 LeetCode

In the course of our research, we identified LeetCode as a potential source of data, owing to its comprehensive repository of coding problems, which are accompanied by solutions that vary in efficiency, measured in terms of time and space complexity. Our initial approach was to systematically extract the problem statements from LeetCode, alongside two distinct solutions for each problem: one serving as a baseline (ground truth) and the other representing the most optimized solution available. This selection criterion was premised on the assumption that a comparative analysis of different solution strategies would yield valuable insights.

However, our efforts to automate the extraction process encountered significant hurdles. LeetCode employs Cloudflare as part of its security measures, which proved to be a formidable barrier against our Selenium based scraping tools. Despite numerous attempts, our automated scripts were consistently thwarted, preventing successful data retrieval.

In response, we explored third-party scrapers specifically designed for LeetCode, namely the tools hosted on GitHub by Bishalsarang (https://github.com/Bishalsarang/Leetcode-Questions-Scraper) and nikhil-ravi (https://github.com/nikhil-ravi/LeetScrape). While these tools promised a workaround to our initial problem, they introduced challenges of their own. For instance, we successfully retrieved question slugs and associated topics using the latter tool but encountered a critical failure when accessing solutions. The tool's reliance on requests.get("https://leetcode.com/api/problems/all/").json() resulted in a JSONDecodeError, indicating a potential change in the API or its accessibility constraints.

Then, our quest led us to a replica website (https://leetcode.ca/all/problems.html) mimicking the LeetCode platform, which presented a seemingly opportune data source without the security of the actual Leetcode platform. Unfortunately, this site predominantly offered single solutions for most problems, lacking the comparative data between least and most optimized solutions we required for our analysis.

Additionally, further investigation revealed that not all problems on LeetCode are associated with multiple solutions. This inconsistency in the availability of comparative data further complicated our attempt to uniformly collect both a ground truth and an optimized solution for each problem. Given these challenges — the robust security measures implemented by LeetCode and the variability in solution availability — we concluded that continuing our efforts to extract data from LeetCode was impractical.

In light of these considerations, we opted to pivot our automated data collection efforts to HackerRank, another platform offering a rich set of coding problems. This decision was driven by the need for a more accessible and consistent source of data for our research objectives.

In addition, rather than directly scraping data from the LeetCode website, we have now identified several GitHub repositories (eg: https://github.com/Akshaya-Amar/LeetCodeSolutions) with either basic or efficient solutions to a small set (about 150) LeetCode questions. From these repositories, we will manually curate pairs of unoptimized and optimized solutions and integrate them into our dataset.

4.2 HackerRank

In our continued exploration of potential data sources for our research, we turned our attention to HackerRank, a platform distinct from LeetCode in several critical aspects, notably its security protocols. Unlike LeetCode, HackerRank does not implement Cloudflare,

which significantly simplified our efforts to automate data extraction. With this advantage, we developed two primary methods for scraping data from the platform.

The initial strategy focused on the extraction of problem statements presented in natural language, in addition to identifying the most optimized solution for each problem. Our initial attempts employed BeautifulSoup for text extraction. However, we encountered a significant challenge: the problem statements on HackerRank often incorporate images, particularly for equations and numerical data, which are not amenable to direct text extraction using BeautifulSoup. To circumvent this obstacle, we innovated an alternative method utilizing Selenium. This approach involved sequentially capturing screenshots of each segment of the problem statement and subsequently amalgamating these images to form a complete visual representation of the problem. To convert these images back into text, we employed EasyOCR. Despite this innovative approach, we faced issues with image quality that led to incomplete or inaccurate text extraction, characterized by the omission of critical tokens from the problem statements.

Confronted with these challenges, we devised a secondary approach that bypassed the extraction of problem statements altogether. Instead, this method concentrated on obtaining two solutions for each problem: a baseline solution (ground truth) and the most optimized solution available on the platform. This shift in strategy stemmed from the difficulties encountered in accurately and efficiently converting problem statements from images to text, highlighting the complexities of dealing with diverse data presentation formats in online coding platforms.

The second strategy focuses particularly on Java7 solutions, chosen for its prevalence among the platform's Java-based submissions. Our aim is to extract two distinct sets of solutions for each problem: the most optimized solution and a baseline solution that, while not optimized for efficiency, is correct.

To identify the most optimized solution, we target the top-ranked solution on HackerRank's leaderboard. This choice is predicated on the assumption that the leading solution, by virtue of its position, has passed all available test cases, thereby demonstrating not only correctness but also optimal performance in terms of time and space complexity.

Conversely, identifying the baseline solution—the least optimized yet correct solution—introduces a unique set of challenges. Our criteria for the baseline solution are that it must rank lowest on the leaderboard while still achieving the same score as the top-ranked solution, ensuring its correctness. The primary obstacle in this endeavor arises from the sheer volume of solutions a single problem may accumulate; a medium-ranked problem, for example, can gather between 20,000 and 30,000 solutions. Given that HackerRank limits the visibility to 100 solutions per page, manually navigating through 200 to 300 pages to locate the least optimized yet correct solution is impractical and inefficient.

To navigate this challenge more effectively, we have conceptualized an innovative solution leveraging a binary search algorithm. This method involves systematically filtering the solutions by specifying Java7 as the programming language, thus significantly reducing the scope of our search and enabling us to identify the desired baseline solution more efficiently. Our team is currently in the process of refining and implementing this binary search approach, with the anticipation that it will streamline our data collection process and enhance the overall efficacy of our research methodology.

Through these experiences, we gained valuable insights into the practical challenges of data extraction in the context of coding problem repositories, informing our ongoing research methodologies and data collection efforts.

GitHub repository: https://github.com/Mayeshamonjur/AISWE_Project

5 USAGE SCENARIO

The development of an AI tool for optimizing Java code using Large Language Models (LLMs) harbors significant potential for real-life applications across various domains. By enabling more efficient and optimized programming practices, this project stands to revolutionize software development processes, enhance educational platforms, and contribute to the efficiency of competitive programming environments. Below, we outline several key areas where our project's outcomes can be directly applied.

5.1 Software Development Efficiency

In the realm of software development, the proposed AI tool can significantly reduce the time and effort required for code optimization. By automating the identification of inefficient code and suggesting optimized alternatives, developers can focus on higher-level design and problem-solving tasks. This could lead to faster development cycles, reduced debugging and maintenance efforts, and overall improved software performance and quality.

5.2 Educational Platforms

Educational platforms that offer programming courses and exercises, such as LeetCode and HackerRank, could integrate our AI tool to provide immediate, intelligent feedback to learners. This would not only help students understand optimization techniques but also encourage the adoption of best coding practices from an early stage. It could revolutionize how programming is taught and learned by making optimization an integral part of the educational process. These competitive programming platforms could also use the AI tool to offer participants insights into how their code might be optimized for better performance and efficiency. This could elevate the competitive landscape by pushing for not only correct solutions but also the most optimized ones, fostering a deeper understanding of algorithmic efficiency and computational complexity among competitors.

5.3 Industry Adoption

Beyond educational and competitive realms, the industry at large stands to benefit from the adoption of such an AI tool. Companies focusing on software development, cloud computing, and data-intensive applications would find immense value in automating code optimization to improve execution speed and resource utilization, directly impacting cost efficiency and system scalability.

In conclusion, the real-life applications of this project are vast and varied, touching upon essential aspects of software development, education, competitive programming, and industry practices. By bridging the gap between AI and programming, the proposed tool

not only enhances Java code optimization but also sets a precedent for future innovations in the field.

REFERENCES

[1] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, and Hugh Leather. 2023. Large Language Models for Compiler Optimization. *arXiv:2309.07062* (2023). https://arxiv.org/abs/2309.07062

[2] Shuai Lu Zhangyin Feng Duyu Tang Shujie Liu Long Zhou Nan Duan Alexey Svyatkovskiy Shengyu Fu Michele Tufano Shao Kun Deng Colin Clement Dawn Drain Neel Sundaresan Jian Yin Daxin Jiang Ming Zhou Daya Guo, Shuo Ren. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. *arXiv:2009.08366* (2020). https://arxiv.org/abs/2009.08366

[3] Martin Monperrus, Zimin Chen, and Sen Fang. 2023. Supersonic: Learning to Generate Source Code Optimizations in C/C++. *arXiv:2309.14846* (2023). https://arxiv.org/abs/2309.14846

[4] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2023. Learning Performance-Improving Code Edits. *arXiv:2302.07867* (2023). https://arxiv.org/abs/2302.07867

[5] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. 2023. Large Language Models as Optimizers. *arXiv:2309.03409* (2023). https://arxiv.org/pdf/2309.03409