2024 5th IEEE Global Conference for Advancement in Technology (GCAT)
Karnataka, India. Oct 4-6, 2024

Code Optimization for Python using Large Pre-Trained Model

Adithya, Gokulan S, Mahati Reddy, Varshith M, Meena Belwal
Department of Computer Science and EngineeringAmrita School of Computing, Bengaluru
Amrita Vishwa Vidyapeetham, India
adithyaprasanna121@gmail.com, sgokuloff@gmail.com, mahati1309@gmail.com, varshith0683@gmail.com, b_meena@blr.amrita.edu

2024 5th IEEE Global Conference for Advancement in Technology (GCAT) | 979-8-3503-7668-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/GCAT62922.2024.10923909

***Abstract*— Optimizing Python code is essential for enhancing performance and efficiency. This project investigates the use of large pre-trained language models, specifically GPT (Generative Pre-trained Transformer), for Python code optimization. By leveraging the advanced capabilities of these models, we aim to improve traditional optimization techniques. Our approach involves preprocessing Python code, feeding it into the pre-trained model, and generating optimized code sequences. Experimental results show significant improvements in code efficiency and execution time, validating the effectiveness of our method. Additionally, we explore the practical implications, challenges, and future directions of incorporating large pre-trained models into code optimization, aiming to bridge the gap between standard optimization techniques and Python programming for a more intelligent and efficient software development practice.**

Keywords—Performance, Efficiency, Large Pre-trained language Models, GPT (Generative Pre-trained Transformer), Preprocessing, Execution time

I. INTRODUCTION

In the digital beat that does not stop growing and where software has infiltrated most of the present day's life fields, the search of efficacy is more recognized than ever. Whether it be a personal project, a commercial product, or a contribution to the open-source community, the goal is to optimize the code for speed, resources and scalability, and because of this, the optimization of code is a common pursuit. Python, now being appreciated for its simplicity, easy-to-read syntax, and an impressive supply of libraries, stands out as a natural choice for the developers who want to transform their thoughts into an actual product. Although at enhanced phases of tasks the need for neat code is fundamental, the more complex and intricate the task is the more imperative of code duplication becomes.

Picture this: yet, you have not even imagined the challenges you will face during the deployment phase as you progressed from designing application architecture to crafting its functionality. It will be a task for the ages and one of the most worthwhile ones as well. So, despite using every tool and assuring everything turns out right, you find out that there is a slow performance whether it is literal physical hard disk access problem or a program not running smoothly, stops the application from gliding smoothly. Given the task of coding optimization and having to do it in the intermediate code area where high-level Python statements fuse with byte-code instructions, you may find yourself in a bit of a limbo.

Generally, the optimizing of intermediate codes have been a big problem to tackle as this task has a combination of intricateness and complexity which even the seasoned developers may not handle well. Traditional optimization processes although naturally useful in solving larger classes of problems may fail to adequately take into count the dynamic characteristics of Python, which among other things include; dynamically typed variables, automatic memory management, and interpreter-based execution model. In the end, programmers are struggling to boost software performance based on the disappointing results that stifle their imagination and overall productivity.

A research conducted by C. Dubach et al. [1] presents a comprehensive exploration of machine learning-driven techniques to enhance software performance across multiple domains. It introduces a novel approach for predicting program speedup without extensive executions, offering flexibility and efficiency by eschewing reliance on benchmark suites. Apart from that an article by Nima Asadi et al. [2], it delves into runtime performance optimization of tree-based models, showcasing significant speed enhancements through innovative strategies like cache-conscious data layout and vectorization. Moreover, the research by T. Jayatilaka et al. [3] tackles the challenge of compiler optimization selection by leveraging machine learning to tailor optimization sequences based on program characteristics, potentially surpassing traditional one-size-fits-all methods. These pioneering methodologies signify a paradigm shift in software performance optimization, promising substantial improvements in efficiency and effectiveness.

Enter the paradigm shift: Intermediate Code Optimization using Large Pre-Trained Models for Python. At the head of this transformative methodology are the integration of advanced ML techniques using powerful pre-trained models aimed at replacing the current code optimization techniques. Think about leveraging the distributed intelligence of these advanced algorithms, which have been trained on different sections of code from various domains, to dig out the deepest questions on Python optimization.

By using this new paradigm, the process of optimization of Python code moves far beyond traditional limitations and heralds the new age of efficiency and effectiveness. The usage of large pre-trained models by developers yields to an acquirement of a rich knowledge reservoir from which they can track the performance problems with unimaginable accuracy and speed. In terms of execution paths streamlining, memory usage optimization, elimination of repetitions and improving algorithms efficiency, the sky is the limit for enhancements.

II. RELATED WORKS

A study by Sameer Kulkarni et al. [4] applies the sequential stage optimisation challenge to the NEAT

algorithm using the Jikes RVM Java JIT compiler. According to the study's findings, the best combination of programming techniques to employ with the optimisation tool was ANN based on NEAT. As a result, the research is different from the traditional programming set of preset stages. The tests yielded notable gains in performance on Java benchmarks: speedups ranging from 6% to 24% were seen in mpegaudio/compress, 8% were obtained in running time improvements from adaptive compilation, and 8.2% were obtained in non- adaptive mode. The method shows how to solve optimisation issues using the NEAT compiler.

An article by Mircea Namolaru et al. [5] presents an approach that systematically uses machine learning to generate numerical features from a code in order to represent intelligent compiler optimisations in the GCC compiler. The authors gave an example of how embedded system designers may benefit from their GCC feature extractor and how effectively it could be integrated with MILEPOST GCC. The use case demonstrates how the technique improves performance and optimises the programme to accelerate execution. The study emphasises the need of selecting programme characteristics that are pertinent to the specific optimisation challenge and advocates for a comprehensive evaluation of the features' quality and relevance. They assert that their method might be useful in the field of compilers and could be used as a benchmark for other methods for predictive precedence operations.

Testing the architecture of assembly codes may be difficult since increasingly modern codes include dynamic and sophisticated instruction sets. The ArCheck approach made possible by Niranjan Hasabnis et al. [6] is a rigorous methodology for comparing relevant assembly codes and comparable IR snippets provided by various compiler code generators, is described by the authors of this publication. the creation of a test strategy with objectives based on IR semantics to deliver better examples, as well as the development of an architecture-neutral method to validate machine code and IR behaviours. Using ArCheck, a comprehensive benchmarking of the GCC's x86 code generator revealed 39 semantically distinct assembly language instructions; all of these problems were resolved with bug fixes.

The study by R. S. Olson et al [7] introduces TPOT, a machine learning tool that implements genetic programming to solve machine learning pipeline optimization problems. It was assessed on a total of 150 supervised classification problems across a variety of specific domains to highlight its remarkable results in comparison to the traditional machine learning techniques. TPOT, which is an abbreviation of "The Python Optimized Tool," automatically builds pipelines that are both precise and brief, exhibiting how it can contribute to the improvement of automated machine learning techniques. The fusion of Genetic Programming with Pareto Optimization has enabled TPOT to search for the most suitable pipelines without the involvement of humans. This is a great step forward in automated machine learning.

The study by Lei Shen et al. [8], centers on the effectiveness of machine learning algorithms in detecting code smells, highlighting design patterns Data Class and Feature Envy. It examines the role of hyper-parameter optimization alongside the four different optimizers and the six classifiers. The performance assessment, which is displayed by AUC, indicates that source code smell detection is remarkably improved by the hyper-parameter optimization. To be specific, Differential Evolution (DE) optimizer is leading the rest when coupled with the random forest classifier. These results also highlight the future improvement in code smell using DE optimizer by changing the parameter optimization.

The research by Huating Wang et al. [9], presents SuperSonic, the RL framework operating as a tool which serves its purpose ease of presentation of the RL in the scenario compilers allowing the promotion of complex optimization By Code. It aims it at underlying diligent handcrafting of the architecture by featuring parametrized constructs and leveraging on deep RL and multi-task learning approaches. Noteworthy of all, SuperSonic is endowed with a meta-optimizer that autonomously singled out the best RL architecture among those that have been trained for specific optimization jobs. The evaluations after four code optimization problems indicate that SuperSonic is able to outperform in 1.75x to hand-tuned methods and it is able to save 50% to 100% of the search in optimization times after implementation due to its ability to expedite the process in deploying such implementations.

The study by Dokkyun Yi and team [10], proposes a new optimization approach to overcome the non-convexness problem regarding the cost function in machine learning in order to enhance the learning process. If the parameter update rule of the ADAM method is made more optimal, it can be expected that the proposed improved would have better convergence towards optimum solutions. The numerical assessments against GD, and the other two methods, ADAM and AdaMax reveal the effectiveness of the method in optimizing non-linearly complex functions and convolutional neural network models. Results are shown to be ahead of traditional methods in every experiment including that of the MNIST and cifar10 datasets, with an improved convergence towards global minima, even with problems that begin with local minima. Hence, these results give us the evidence that the suggested method can be useful in the future and advance the optimization processes of machine learning applications.

The article by A. Kaur et al. [11] is presented a new combined algorithm, SP-J48, that joins the Sandpiper Optimization Algorithm (SPOA) with pruned-B machine learning approach for efficient code smells detection in software systems. SPOA is an algorithm that was motivated by the behavior of sandpipers and showed higher performance on test functions on benchmarks compared to other algorithms also. In order to enable the detection of inefficient code smells for enhanced software maintenance, SP-J48 puts weight on the detection and classification of various code smells that include blob, feature envy, data class, functional decomposition, and spaghetti code. Our findings confirm the superiority of SP-J48 in terms of certain metrics parameters, such as precision, recall and average number of defects, and they also prove that such method is efficient in improving object-oriented software cohesion.

In order to bridge the existing gap in natural language to machine code translation, the research by Shashank Sridhar et al. [12] analyses a model that translates natural language into C code. By utilizing Flask and Python, this attempts to allow the building of large-scale applications of limited scope and independent of certain platforms possible. Similar methods and models from earlier research, which

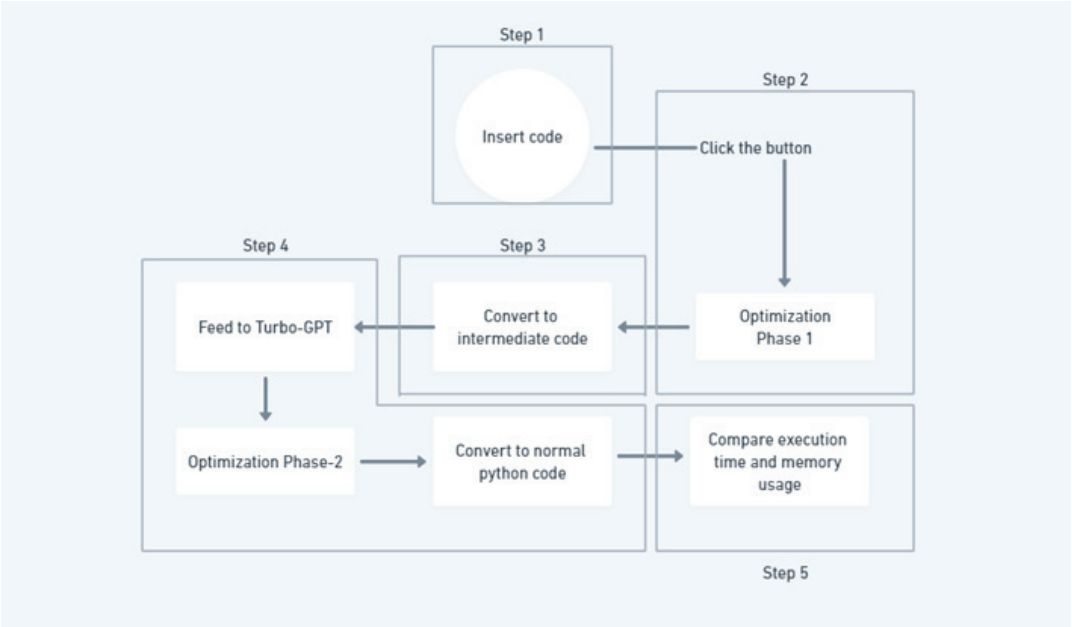

Fig. 1. Flowchart of the Proposed Model

concentrated on the impact of neural networks and genetic algorithms in particular on performance enhancement, are employed in this study. The study emphasises the necessity of compiler feedback and syntax validation to guarantee correct and appropriate translation of natural language into code. It also demonstrates the value of compiler methods in improving code quality.

The research by Fengqian Li et al. [13], focuses on building optimization procedures by exploiting static information obtained through intermediate representation trees traversal. It proposes template-based and multi-phase architectures toenhance the distinctive feature extraction and prediction correctness. The system is to be evaluated by competing the object files running speed over the GCC O3 optimization levels, which results in averaging 5% speedup. The approach illustrates its efficiency by implementing a model that optimizes different strategies coupled with KPI-ready benchmarks. This technique illuminates feature extraction before achieving milestones in GCC, and reveals the necessityof utilizing both static and dynamic features for holisticprogram profiling and optimization.

The study by Abid M Malik [14] describes an architecture exploiting spatial in DFGs so as to optimize compiler through machine learning. From spatial data, it picks features that assist it in compiler optimization and it outperforms the IBM Milepost in option selection, resulting in notable performance improvements especially for Decision Tree and Support Vector Machines. The paper focuses on compiler optimizations' levels, feature generations, and machine learning techniques through which the paper revealed the role of spatial based features in improving the execution times, in addition to the role of machine learning in compiler optimization.

CodeZero, an artificial balancing agent that has been trained on huge data sets to immediately devise the best optimization algorithms for each individual application in just one trial is the topic of the article, authored by J. Wu et al. [15]. The hurdle of long and sluggish optimization search processes is what CodeZero tackles with the assistance of deep reinforcement learning algorithms, the compiler environment sample-efficient learning in the world model. The evaluation of CodeZero on benchmark suites and production-level optimization problems by engine shows its leading performance features and zero- shot capabilities which outscores manually created optimization models by experts.

III. METHODOLOGY

The proposed model is depicted in Figure 1 and this section is about how the steps mentioned in the model are followed. Firstly, the application starts by opening and asking the user to input the directory in which the Python file to be optimized is located. This is due to the following steps we take in the course of the enhancement of the code. Once the user enters the file path, or selects it from directory through the set path text box, the user is able to click on a button marked ‘Optimize’. To begin with it, the application introduces basic modifications namely reformatting indentations and eradicating lines. It then compiles the code into a form which is in an in-between form, and then applies complex optimization to the code. Last but not least, the interfaced optimized intermediate code is passed to the GPT-4 model via the model’s API for minor aesthetic adjustments before translation to Python. Once the optimization is done, the new optimized Python code is saved with the same name, and is in the same directory as the input data file so that comparisons can be easily made between the original and the optimized. This makes the work flow efficient as it is user friendly while at the same time utilizing complex optimizers to enhance python coding.

A. Code Insertion into Application

During methodology stage one of our project, we ensure that the code file of Python is incorporated perfectly into the application. This is used to ensure direct compatibility and integration of our interface or application with the rest of the system. After that, our step is to make phase-1 optimizations, for example, applying the correct code indentations, removing excessive blank lines and optimizing the comments. Also, we make enhancements on loops to make them more efficient, thus improving the quality of the code.

It is beneficial to take the time to perform all of these operations conscientiously because it prepares the code further for future improvements as the project moves through the development cycle.

B. Optimization Phase 1

Some of the recommended steps in the process of phase-1 optimizations include optimization looks into a number of aspects in an attempt to improve the code quality, readability and efficiency. Initially, the code is preprocessed as per specific style templates, so that it becomes easy to understand and adhere to a certain style. Comments within the code are then uniformly deleted to increase the organization of the script, furthermore all unnecessary blank lines are deleted to reduce the clutter.

As part of the elimination of unnecessary overhead and to meet the objective of huge code compactness, functional functions are converted to lambda functions wherever possible. Also, classical for and while cycles are replaced by list comprehensions or generator expressions, improving efficiency and decreasing the code density. This transformation also aids in making the logic easier and enhances the execution speed than the equivalent originating languages which are optimized according to the options for the best practices.

Last but not least, the main part of the code unites these optimizations while checking the correct work and needed parameters. This section represents the central management of the optimization process; here, the sequence of transformations is carried out in the manner that would produce an enhanced and more optimal version of the original code.

By following these fundamental codes optimization, the code works to improve the qualities of high, readability, and maintainability of the codebase in the future when expanded.

C. Optimized code to Intermediate Code

In the third step of our project, we further go down the level of optimization to the next level by converting the initially optimized pure Python code to a form called the intermediate representation. It is acting as the middle ground between the Python code's syntax and the machine code. During this process, we change our perspective and start focusing on especially, repairing and optimizing the code, going into detail about small aspects, algorithmic efficiency, data structure usage, or just peculiarities of the language's syntax and idioms. We use specialized optimization methods adapted to the characteristics and needs of our project so as to obtain the highest possible levels of performance increase and at the same time without compromising the factors of code comprehensibility and sustainability. Thus, being careful with the interpretation of the intermediate code, we create conditions for applying more complex optimization techniques at the next stages of our work, so as to achieve our goal in the form of a final solution, which would be optimal in terms of a balance between computational efficiency and code aesthetics.

D. Optimization Phase 2 (via Pre-Trained Model)

In the fourth part of our methodology, we make use of the mode's API to input to it the phase-1 optimized intermediate code created in the third phase. With the help of this advanced language model, let it try to go further for the optimization of solutions and improvements. Therefore, by running the code for the program to be executed by Turbo-GPT, the prompt gains access to this vast knowledge base as well as the advanced understanding of language. The model creates corrections, improvements, and options for the reformulating the given code snippet based on its evaluation. These suggestions are focused on increasing not just the speed but also the efficiency and beauty of the code.

After getting the suggestions from pre-trained model, we proceed to analyze them and, if appropriate, carry the essential optimizations to the intermediate code. The code structure is gradually fine-tuned, algorithms are made refined and, generally, new approaches to solving a problem at hand are experimented upon. While making these optimizations, we always try to stay as close as possible to the code intent and logic and, at the same time, improve the code's performance and usability as much as possible.

After that, we convert the refined intermediate code with the help of insights shared by Turbo-GPT back to Python code. It is for this optimized Python code that we have all put in our final best effort culminating in this piece of highly advanced yet logical and synthetic effort with the help of AI.

E. Evaluation of the Application

The last step in the final project methodology is to define the efficiency comparison of the unoptimized code and the end optimized one. This assessment focuses on two key metrics: storage performance and processing time of the memory. In this way, through the series or tests, we specify the results concerning how the optimizations influence the utilization of the code's resources. Memory profiling is done to check whether the optimizations make the program use more resources than it used to; runtime tells the program's speed and how fast it executes a command. This qualitative analysis allows us to estimate the material advantages of optimization, showing the changes in the quantitative characteristics based on our systematic approach. The outcomes of this phase are significant as they give qualitative and quantitative information about the benefits which support the usage of basic source code upgrades, intermediate optimizations, and advanced AI based updates in the software development life cycle.

IV. RESULTS AND ANALYSIS

This is followed by the results and analysis section in the project that helps in observing execution time and memory utilization of various test cases. These test cases contain the Fibonacci series, the problem of the 3sum, finding the length of the non-repeating substring, and the check of whether it contains a pair summing to a given value. Every test case can be viewed as another specific computational problem that let us evaluate the performance of our optimization method under different problems. The result of our analysis is outlined in two specific tables where the results are presented in a very descriptive manner strictly estimating the efficiency of the developed optimization method. To quantify the improvements or gains that were made in optimizing this particular aspect, it is now possible to assert, to a certain level of accuracy, the rate of execution time and memory consumption before and after the optimization process were measured.

Table 1 shows the comparison between the original code and the code after optimization for the four test examples.

The result analysis shows a significant degree of differences by comparing the optimized code with the pre-optimized code by proving that the time complexity of the optimized code is less than the pre-optimized code, which is in between 20% to 65%.

TABLE I. COMPARISON OF EXECUTION TIME

Test Cases	Un-Optimized Code (sec)	Optimized Code (sec)	Improvement (%)
1 (Fibonacci)	0.0027	0.0020	25.92
2 (3sum)	0.0031	0.0012	61.29
3 (length of substring)	0.0014	0.0010	28.57
4 (Pair Summing)	0.0024	0.0011	54.16

TABLE II. COMPARISON OF MEMORY USAGE

Test Cases	Un-Optimized Code (MB)	Optimized Code (MB)	Improvement (%)
1 (Fibonacci)	0.0710	0.0637	10.34
2 (3sum)	0.1114	0.1032	07.30
3 (length of substring)	0.1215	0.0952	21.65
4 (Pair Summing)	0.1261	0.1246	01.24

This significant enhancement further cements the effectiveness of the optimization strategies proposed in this work as key factors for faster execution of code and better computational performance.

Table 2 looks at the memory usage of the same test cases, but with both versions of the code: unoptimized and optimized. They include relative memory consumption reduction that varies from 1% to 22%. This decrease in memory also shows the good employment of the system resources in managing memory space and hence emphasizes the practical usefulness of our model.

The first general limitation that can be pointed out with regard to the proposed model is the lack of possibility to enhance critically massive functions. The Turbo-GPT model that we employ has a certain token limit, to the maximum of which we did not take the code snippets; nonetheless, it proves to be less efficient in dealing with large corpora. This restriction prevents us from effectively performing further enhancement on large or composite Python scripts; Hence, our methodology only finds relevancy on compact or mid-scaled python scripts. To overcome this, one must either raise the token limit of the chosen model or search for other ways of analyzing large code pieces, making it possible to encompass additional optimization for more massive endeavors.

These comparisons provide convincing proof of the applicability of the described optimization technique. The comparison of the results of enhancing the execution time and the memory space for different test cases confirmed the viability of presented model and provided the possibility for improving the performance of the model and usage of the resources. From such findings, it did not only demonstrate that our approach apt for real-world problems but it also highlighted that our work can enhance the current manner of developing software. Thus, considering certain specific computational issues, it can be stated that the proposed approach appears to be rather effective and is applicable within a vast array of Python programs. Notably, there is a drawback in the number of tokens that Turbo-GPT model can process; something that makes it very hard to optimize long code sections or large-scale applications, programs, etc. Nevertheless, the methodology is proved to be rather efficient for scripts of average size, from medium to small though.

V. CONCLUSION AND FUTURE SCOPE

Therefore, with certainty, it is possible to assert that our work covers and implements an effective approach to discussing and implementing improvements and transformations to Python code through the creation of large scale pre-trained language models, particularly GPT. After carrying out phase-1 level or even simple optimizations and intermediate or higher-level code transformation along with the use of AI optimization techniques to illustrate the level of performance optimization realized alongside the effective use of available resources.

The adaptation of the Turbo-GPT in the described pipeline optimization has brought tangible benefits where improvements in terms of execution time and memory space are concerned as evident from the optimized version of Python solutions of various problems of numerical series and matrices to different test cases as test from the Fibonacci series to 3-sum problem. Across the test cases, the amount of time taken was reduced by 20-65% and the amount of memory consumed was reduced by 1-22%. Although there are several exciting outcomes, it has a significant drawback as the model of Turbo-GPT is limited by the token and, therefore, optimizes only small or middle-sized fragments of code and is unable to optimize large or more complex code segments because of this limitation.

These findings shed light into the fact that large PLMs can indeed be utilized for code optimization, opening up the research domain towards enhancing the performances of existing softwares and the efficient utilization of available resources. This research stands midway between what has been traditionally considered as optimization and the concept of the new age AI optimization techniques, and tries to provide an approach that proposes to create a better system.

To further our work in the future, several useful directions can be continued. First, generalizing the applicability of the proposed research to more types of structures and paradigms, existing within the frameworks of the Python language, could optimize the impact of the software in various domains. Also, the inclusion of feedback mechanisms as user constraints or performance indices in the optimization may complicate the optimizations further but the derived optimizations are specific to the project.

Exploring how to apply the herein proposed optimization tools into cloud computing or distributed framework could potentially enable the large scale and parallel optimization to help software development. However, possible improvements for future works regarding a refined fine-tuning of optimization algorithms based on AI and the creation and implementation of improved and larger pre-models could additionally increase the performance and generalization of the current studies applied in a more complex optimization problem for software development.

Therefore, our study complements the prior works aimed at the development of the principles for the improvement of

the presented beforehand approaches that contribute to the advancement of the current state of techniques in code optimization and can support the preferable creation of high-speed application software.

REFERENCES

[1] C. Dubach, J. Cavazos, B. Franke, Grigori Fursin, Michael F.P. O'Boyle, and Olivier Temam, “Fast compiler optimisation evaluation using code-feature based performance prediction,” May 2007, doi: https://doi.org/10.1145/1242531.1242553.

[2] N. Asadi, J. Lin and A. P. de Vries, "Runtime Optimizations for Tree-Based Machine Learning Models," in IEEE Transactions on Knowledge and Data Engineering, vol. 26, no. 9, pp. 2281-2292, Sept. 2014, doi: 10.1109/TKDE.2013.73.

[3] T. Jayatilaka, H. Ueno, G. Georgakoudis, E. Park, and J. Doerfert, “Towards Compile-Time-Reducing Compiler Optimization Selection via Machine Learning,” 50th International Conference on Parallel Processing Workshop, Aug. 2021, doi: https://doi.org/10.1145/3458744.3473355

[4] S. Kulkarni and J. Cavazos, “Mitigating the compiler optimization phase-ordering problem using machine learning,” CiteSeer X (The Pennsylvania State University), Oct. 2012, doi: https://doi.org/10.1145/2384616.2384628.

[5] Mircea Namolaru, A. Cohen, Grigori Fursin, A. Zaks, and A. Freund, “Practical aggregation of semantical program properties for machine learning based optimization,” HAL (Le Centre pour la Communication Scientifique Directe), Oct. 2010, doi: https://doi.org/10.1145/1878921.1878951.

[6] N. Hasabnis, R. Qiao and R. Sekar, "Checking correctness of code generator architecture specifications," 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), San Francisco, CA, USA, 2015, pp. 167-178, doi: 10.1109/CGO.2015.7054197. keywords: {Generators;Testing;Assembly;Semantics;Silicon;Computer bugs;Registers},

[7] R. S. Olson and J. H. Moore, “TPOT: A Tree-based Pipeline Optimization Tool for Automating Machine Learning,” proceedings.mlr.press, Dec. 04, 2016.

[8] L. Shen, W. Liu, X. Chen, Q. Gu and X. Liu, "Improving Machine Learning-Based Code Smell Detection via Hyper-Parameter Optimization," 2020 27th Asia-Pacific Software Engineering Conference (APSEC), Singapore, Singapore, 2020, pp. 276-285, doi: 10.1109/APSEC51365.2020.00036.

[9] H. Wang et al., “Automating reinforcement learning architecture design for code optimization,” Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction, Mar. 2022, doi: https://doi.org/10.1145/3497776.3517769

[10] D. Yi, J. Ahn, and S. Ji, “An Effective Optimization Method for Machine Learning Based on ADAM,” Applied Sciences, vol. 10, no. 3, p. 1073, Feb. 2020, doi: https://doi.org/10.3390/app10031073.

[11] A. Kaur, S. Jain, and S. Goel, “SP-J48: a novel optimization and machine-learning-based approach for solving complex problems: special application in software engineering for detecting code smells,” Neural Computing and Applications, vol. 32, no. 11, pp. 7009–7027, Apr. 2019, doi: https://doi.org/10.1007/s00521-019-04175-z

[12] S. Sridhar and S. Sanagavarapu, "A Compiler-based Approach for Natural Language to Code Conversion," 2020 3rd International Conference on Computer and Informatics Engineering (IC2IE), Yogyakarta, Indonesia, 2020, pp. 1-6, doi: 10.1109/IC2IE50715.2020.9274674.

[13] F. Li, F. Tang and Y. Shen, "Feature Mining for Machine Learning Based Compilation Optimization," 2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, Birmingham, UK, 2014, pp. 207-214, doi: 10.1109/IMIS.2014.26.

[14] A. M. Malik, "Spatial Based Feature Generation for Machine Learning Based Optimization Compilation," 2010 Ninth International Conference on Machine Learning and Applications, Washington, DC, USA, 2010, pp. 925-930, doi: 10.1109/ICMLA.2010.147.

[15] J. Wu, C. Deng, J. Wang, and M. Long, “Supercompiler Code Optimization with Zero-Shot Reinforcement Learning,” arXiv.org, Apr. 24, 2024. https://arxiv.org/abs/2404.16077 (accessed May 01, 2024)

[16] Pecheti, Shiva Teja, H. M. Basavadeepthi, Nithin Kodurupaka, and Meena Belwal. "Recursive Descent Parser for Abstract Syntax Tree Visualization of Mathematical Expressions." In 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), pp. 1-6. IEEE, 2023.

[17] Vishwas, Gade, Nayini Sai Nithin, Peddineni Varshith, and Meena Belwal. "Unveiling the World of Code Obfuscation: A Comprehensive Survey." In 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), pp. 1-8. IEEE, 2023.

[18] Gurusamy, Bharathi Mohan & Kumar, Prasanna & Srinivasan, Parathasarathy & Kb, Hanish & Pavithira, "Text Summarization for Big Data Analytics: A Comprehensive Review of GPT 2 and BERT Approaches", Springer (Accepted), 2023

[19] Jose, A., Harikumar, S. (2022). COVID-19 Semantic Search Engine Using Sentence-Transformer Models. In: Raman, I., Ganesan, P., Sureshkumar, V., Ranganathan, L. (eds) Computational Intelligence, Cyber Security and Computational Models. Recent Trends in Computational Models, Intelligent and Secure Systems. ICC3 2021. Communications in Computer and Information Science, vol 1631. Springer